

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Strojnícka fakulta

Evidenčné číslo: SjF-5226-87740

PressureShield: miniatúrny prístroj na riadenie tlaku v nádobe

Diplomová práca

2021

Bc. Martin Staroň

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Strojnícka fakulta

Evidenčné číslo: SjF-5226-87740

**PressureShield: miniatúrny prístroj na riadenie
tlaku v nádobe**

Diplomová práca

Študijný program: automatizácia a informatizácia strojov a procesov

Študijný odbor: kybernetika

Školiace pracovisko: Ústav automatizácie, merania a aplikovanej informatiky

Vedúci záverečnej práce: prof. Ing. Gergely Takács, PhD.

Konzultant: Ing. Erik Mikuláš

Bratislava 2021

Bc. Martin Staroň



ZADANIE DIPLOMOVEJ PRÁCE

Študent: **Bc. Martin Staroň**

ID študenta: 87740

Študijný program: automatizácia a informatizácia strojov a procesov

Študijný odbor: kybernetika

Vedúci práce: prof. Ing. Gergely Takács, PhD.

Konzultant: Ing. Erik Mikuláš

Miesto vypracovania: ÚAMAI SjF STU v Bratislave

Názov práce: **PressureShield: miniatúrny prístroj na riadenie tlaku v nádobe**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Úlohou študenta je navrhnuť a vyrobiť miniaturizovaný experimentálny modul na riadenie tlaku v nádobe. Prístroj bude slúžiť na výučbu technických disciplín v automatizácii ako je mechatronika, teória riadenia, identifikácia sústav a spracovanie signálov. Prístroj bude kompatibilný s elektronickým rozložením mikroradičovej prototypizačnej dosky Arduino R3. Študent vytvorí programátorské rozhranie v jazyku C/C++ pre prostredie Arduino IDE, pre MATLAB, pre Simulink, prípadne iné programovacie jazyky a vývojové prostredia. Ďalšou úlohou je matematicko-fyzikálna analýza dynamického procesu a následná experimentálna identifikácia modelu. Študent taktiež vytvorí rôzne didaktické inštruktážne príklady na spätnoväzobné riadenie tohto systému: PID riadenie, lineárno-kvadratické riadenie (LQ), prediktívne riadenie (angl. model predictive control, MPC), prípadne iné metódy riadenia.

V rámci diplomovej práce študent musí

- navrhnuť experimentálne zariadenie, vybrať vhodné elektronické a mechanické komponenty, navrhnuť elektrické zapojenie a dosku plošných spojov, navrhnuť mechanické osadenie, vyrobiť a otestovať funkčnosť navrhnutého modulu;
- napísať programátorské rozhranie (angl. application programming interface, API) na ovládanie zariadenia pre rôzne programovacie jazyky a vývojové prostredia (C/C++ pre Arduino IDE, MATLAB, Simulink, príp. iné);
- opísať dynamický jav matematicko-fyzikálnou analýzou, navrhnuť vhodný testovací signál a na základe meraných výsledkov identifikovať neznáme parametre modelu;
- vytvoriť didaktické úlohy spätnoväzobného riadenia metódami PID, LQ a MPC riadenia pre každé vytvorené prostredie.
- využiť prostredie GitHub na manažovanie verzií softvéru a na integráciu do základnej knižnice.

Rozsah práce: 50-70 s.

Riešenie zadania práce od: 15. 02. 2021

Dátum odovzdania práce: 28. 05. 2021

Bc. Martin Staroň

študent

prof. Ing. Cyril Belavý, CSc.

vedúci pracoviska

prof. Ing. Cyril Belavý, CSc.

garant študijného programu

Čestné prehlásenie

Vyhlasujem, že som záverečnú prácu vypracoval samostatne s použitím uvedenej literatúry.

Bratislava, 28. mája 2021

.....
Vlastnoručný podpis

V prvom rade by som sa chcel podakovať vedúcemu práce, prof. Ing. Gergelymu Takácsovi, PhD. za jeho odbornú pomoc, cenné rady pri návrhu hardvéru a softvéru, ochotu pri riešení problémov a za jeho pozitívny prístup, ktorým mi uľahčil tvorbu tejto práce. Ďalej by som sa chcel podakovať Ing. Erikovi Mikulášovi za výpomoc pri navrhovaní hardvéru a za jeho pomoc s návrhom a tlačou 3D komponentov, a Bc. Mgr. Anne Vargovej za rady a pripomienky pri identifikácii, modelovaní a simulovaní.

Bratislava, 28. mája 2021

Bc. Martin Staroň

Názov práce: PressureShield: miniatúrny prístroj na riadenie tlaku v nádobe

Kľúčové slová: PressureShield, AutomationShield, spätnoväzbové riadenie, PID regulátor, LQ regulátor

Abstrakt: Táto práca sa zaoberá experimentálnou doskou PressureShield, ktorá slúži na demonštráciu spätnoväzbového riadenia tlaku v pretlakovej nádobe pomocou vývojovej dosky Arduino. Na začiatku je uvedený prehľad projektov zaoberajúcich sa riadením tlaku, je predstavená iniciatíva AutomationShield a zariadenie, ktoré bolo predchodom dosky PressureShield. Opisuje schému hardvéru, návrh dosky plošných spojov a jednotlivé komponenty, ktoré sa využívajú pri kompletovaní. V rámci práce sú podrobne opísané vytvorené programátorské rozhrania, konkrétnie rozhrania pre prostredie Arduino IDE, MATLAB a Simulink. Tieto programátorské rozhrania sú ďalej využité pri tvorbe príkladov spätnoväzbového riadenia pre každé prostredie, konkrétnie pri tvorbe PID regulátora a LQ regulátora.

Title: PressureShield: A miniature device to control the pressure in a vessel

Keywords: PressureShield, AutomationShield, feedback control, PID regulator, LQ regulator

Abstract: This thesis is dedicated to the design of the experimental PressureShield Arduino expansion module, which serves for the didactic demonstration of feedback control; tracking the pressure in a vessel. First, a literature review of works dealing with pressure control is provided, then the AutomationShield initiative is presented, along with a device that is a predecessor to the PressureShield board discussed here. Furthermore, the thesis gives a description of the electronic circuit, printed circuit board design and individual components used for the final assembly. The next chapter of the thesis details application programming interfaces, more specifically, libraries written for the Arduino IDE, MATLAB and Simulink. These programming interfaces are demonstrated by examples in feedback control for each interface, specifically, realize a PID and an LQ regulated pressure control.

Obsah

Úvod	1
1 Motivácia	2
1.1 Prehľad literatúry	2
1.2 AutomationShield	4
1.3 PressureShield R1	4
2 Návrh hardware	7
2.1 Schéma a návrh PCB	7
2.2 Senzor tlaku	9
2.3 Pneumatické čerpadlo	10
2.4 3D komponenty	10
2.5 Montáž	13
2.6 Zoznam komponentov a cena	13
3 Programátorské rozhrania	15
3.1 AutomationShield API	15
3.2 PressureShield rozhranie pre Arduino IDE	16
3.2.1 Čo je Arduino IDE?	16
3.2.2 PressureShield API	16
3.3 PressureShield rozhranie pre MATLAB	20
3.3.1 Čo je MATLAB?	20
3.3.2 PressureShield API	20
3.4 PressureShield rozhranie pre Simulink	23
3.4.1 Čo je Simulink?	23
3.4.2 PressureShield API	24
4 Príklady spätnoväzbového riadenia	28
4.1 Identifikácia a modelovanie	28
4.2 Nelinearita systému	30
4.3 PID riadenie	33
4.3.1 PID riadenie pre Arduino IDE	34
4.3.2 PID riadenie pre MATLAB	35
4.3.3 PID riadenie pre Simulink	37
4.4 LQ riadenie	38
4.4.1 LQ simulácia	40

4.4.2	LQ riadenie pre Arduino IDE	41
4.4.3	LQ riadenie pre MATLAB	43
4.4.4	LQ riadenie pre Simulink	46
4.5	MPC riadenie	48
4.5.1	MPC simulácia	49
4.5.2	MPC riadenie pre MATLAB	50
5	Záver	52
Literatúra		54
Dodatok A	Arduino IDE kód	56
Dodatok B	MATLAB kód	67

Zoznam obrázkov

1.1	AutomationShield moduly	5
2.1	PressureShield.	7
2.2	Schéma zapojenia PressureShield.	8
2.3	Návrh dosky plošných spojov	9
2.4	Senzor na meranie tlaku a teploty BMP280.	10
2.5	Pneumatické čerpadlo SC-3101PM.	10
2.6	Návrh 3D komponentov.	11
2.7	Pribehy úniku tlaku v jednotlivých nádobách.	12
2.8	Úpravy pri osadení.	13
3.1	Simulink blok Actuator Write.	24
3.2	Simulink blok Reference Read.	25
3.3	Prepisovanie registrov v Simulinku.	25
3.4	Načítavanie hodnôt registrov v Simulinku.	26
3.5	Simulink blok Sensor Read.	26
3.6	Simulink blok PressureShield.	27
4.1	Priebeh vstupu a výstupu identifikácie v čase.	30
4.2	Ustálené hodnoty tlaku pri stabilnej hodnote PWM.	31
4.3	Priebeh natlakovania a odtlakovania nádoby.	32
4.4	Časové konštanty tlaku.	32
4.5	Priebeh vstupu a výstupu PID regulátora v Arduino IDE.	35
4.6	Priebeh vstupu a výstupu PID regulátora v MATLABe.	37
4.7	Simulink PID regulátor.	37
4.8	Priebeh vstupu a výstupu PID regulátora v Simulinku.	38
4.9	Priebeh LQ simulácie.	41
4.10	Priebeh vstupu a výstupu LQ regulátora v Arduino IDE.	43
4.11	Priebeh vstupu a výstupu LQ regulátora bez kompenzácie.	44
4.12	Priebeh vstupu a výstupu LQ regulátora v MATLABe.	45
4.13	Simulink LQ regulátor.	46
4.14	Priebeh vstupu a výstupu LQ regulátora v Simulinku.	47
4.15	Priebeh MPC simulácie.	49
4.16	Priebeh vstupu a výstupu MPC regulátora v MATLABe.	51

Zoznam skratiek

API	Rozhranie pre programovanie aplikácií (z angl. application programming interface)
AREF	Analógová referencia (z angl. analog reference)
CI	Priebežná integrácia (z angl. continuous integration)
DC	Jednosmerný prúd (z angl. direct current)
GUI	Grafické používateľské rozhranie (z angl. graphical user interface)
I2C	Dvojžilová obojsmerná zbernica (z angl. inter-integrated circuit)
LSB	Najmenej významný bit (z angl. least significant bit)
LQ	Lineárno-kvadratický (z angl. linear quadratic)
MSB	Najviac významný bit (z angl. most significant bit)
NTC	Negatívny tepelný koeficient (z angl. negative temperature coefficient)
PCB	Tlačená obvodová doska (z angl. printed circuit board)
PID	Proporcionálno-integračno-derivačný (z angl. proportional-integral-derivative)
PWM	Impulzová šírková modulácia (z angl. pulse width modulation)
SCL	Sériový časovač (z angl. serial clock)
SDA	Sériové dátá (z angl. serial data)
SISO	Jeden vstup, jeden výstup (z angl. single-input, single-output)
SMD	Zariadenie na povrchovú montáž (z angl. surface mount device)
SPI	synchrónne sériové periférne rozhranie (z angl. serial peripheral interface)

Zoznam fyzikálnych veličín

Symbol	Veličina	Jednotka (iná jednotka)
A	matica dynamiky v CT stavovom priestore	-
B	matica vstupov v CT stavovom priestore	-
C	elektrická kapacita	F (μ F)
C	matica výstupov v CT stavovom priestore	-
D	matica priamej väzby vstupu na výstup v CT stavovom priestore	-
<i>e</i>	regulačná odchýlka	-
G	Matica MPC účelovej funkcie, pričom platí $g^T = x_k^T G^T$	-
H	Hessián, Hessova matica	-
<i>I</i>	elektrický prúd	A (mA)
<i>k</i>	konštantá modelu	-
<i>K_D</i>	derivačná konštanta PID regulátora	-
<i>K_I</i>	integračná konštanta PID regulátora	-
<i>K_P</i>	proporčná konštanta PID regulátora	-
<i>p</i>	tlak	Pa (hPa)
Q	penalizačná matica stavov	-
<i>R</i>	elektrický odpor	Ω
R	penalizačná matica vstupov	-
<i>t</i>	čas	s (ms)
<i>u</i>	vstupný signál	-
<i>U</i>	elektrické napätie	V
<i>y</i>	meraný výstupný signál	-

Úvod

S postupným vývojom technológií je potrebné súbežne vyvíjať aj riadiace procesy, ktoré nám umožňujú využívať najnovšie výdobytky modernej technológie v ich plnom potenciály. Vo všetkých technických oblastiach sa časom dostanú procesy do bodu, kedy už nestaciť riadiť ich pomocou ľudského faktoru, ale je potrebné zapojiť do systému technické riadenie, ktoré napomôže zvýšiť presnosť a efektivitu riadenia. Na tento účel slúži vo svete automatizácia. V dnešných dňoch sa vo svete využíva automatizácia vo väčšine výrobných procesov, no taktiež sa pomocou nej vo veľkom riadia procesy, ktoré nás obklopujú v bežnom živote.

Príkladom riadenia procesov pomocou automatizácie v priemysle ale aj v bežnom živote je spätnoväzbové riadenie. Nezáleží na tom, či sa jedná o riadenie v pneumatickom okruhu výrobnej haly pomocou výkonu kompresora, alebo o udržanie nastavenej teploty v miestnosti domácnosti. Systémy, ktoré si vedia upraviť hodnotu vstupnej veličiny vďaka sledovaniu výstupu zo systému a jeho porovnaniu s požadovanou hodnotou vo veľkej miere prispievajú ku zdokonaľovaniu technológií.

Aby sa však mohol pohybovať vývoj automatizácie zariadení dopredu je potrebné prispôsobiť tomuto pokroku aj experimentálne a výučbové procesy, ktoré napomáhajú študentom a vedcom v ich pokroku v danej oblasti. Z tohto dôvodu potrebujú mať k dispozícii laboratórne pokusné zariadenia, na ktorých by bolo možné experimentovať s konkrétnou problematikou. Takéto laboratórne zariadenia však môžu stať niekoľko desiatok tisíc eur, a ich rozmery môžu byť príliš veľké, čo môže vo veľa prípadoch znamenať ich nedostupnosť pre ľudí, ktorí by ich potrebovali na testovanie a edukatívne účely. S myšlienkovou sprístupnenia zariadení na demonštráciu automatizácie a spätnoväzbového riadenia bola vytvorená iniciatíva AutomationShield, ktorej cieľom je vývoj zariadení malých rozmerov, zostrojených s čo najmenšími finančnými nákladmi.

V rámci tejto iniciatívy sa v mojej práci bude postupne riešiť návrh hardvéru a softvéru nového zariadenia slúžiaceho na výskum systémov so spätnoväzbovým riadením, konkrétnie sa jedná o systém, v ktorom sa reguluje hodnota tlaku v pretlakovej nádobe.

1 Motivácia

Ako je napísané v úvode, vývoj automatizácie a zariadení ovládaných spätnoväzbovým riadením je stále aktuálnejšia téma. Z tohto dôvodu som sa aj ja chcel podrobnejšie oboznámiť s touto problematikou. Okrem cieľa dozvedieť sa viac o samotnej spätnoväzbovej regulácii a o typoch regulátorov, ktoré sa v tejto oblasti používajú, chcel som mať možnosť navrhnúť experimentálne zariadenie. Ďalším cieľom bolo navrhnúť pre toto zariadenie API, s ktorého využitím by som mohol naprogramovať samotnú spätnoväzbovú reguláciu. Zároveň ma však lákala myšlienka prispieť ku vývoju v tejto oblasti a zstrojiť niečo, čo by mohlo pomôcť k vzdeleniu ďalších ľudí. S týmito cieľmi vznikla táto práca, ktorá sa zaoberá spätnoväzbovým riadením tlaku v pretlakovej nádobe pomocou platformy Arduino.

1.1 Prehľad literatúry

Skôr, ako som začal riešiť samotný vývoj môjho zariadenia, venoval som určitý čas prehľadu projektov vo svete, ktoré majú niečo spoločné buď so spätnoväzbovým riadením, s reguláciou tlaku, alebo s únikom tlaku z pretlakových nádob. V tejto podkapitole je uvedený prehľad niektorých článkov a projektov, ktoré sa zaoberali či už automatizáciou, alebo konkrétnou problematikou riadenia natlakovania uzavretej nádoby pomocou komprezora.

V článku [1] využili autori PID regulátor na riadenie systému s DC motorom pomocou vývojovej dosky Arduino Uno. Na riadenie motora využili PWM signál, kde sledovali odozvu systému pri rôznych hodnotách PWM. Neskôr zakomponovali do riadenia PID reguláciu. Pokusmi sa im podarilo dokázať možnosť ovládať a stabilizovať otáčky motora zahrnutého v systéme pomocou PID regulácie riadenej Arduinom. Tento fakt si overili na viacerých referenčných hodnotách.. Konštanty regulátora získali pomocou metódy pokus a omyl.

V [7] Esteves a kol. využili LQ regulátor na riadenie nízkonákladovej kvadrokoptéry riadenej doskou Arduino na stabilizáciu letu a letovej hladiny. Najskôr si vytvorili model, pomocou ktorého matematicky opísali celý systém, ktorý následne využili na odhad matíc Q a R. Vytvorené matice potom využili pre simuláciu riadenia náklonu a výšky kvadrokoptéry, pričom na odhad stavov využili Kalmanov filter. Po úspešnej simulácii úspešne otestovali riadenie pomocou LQ regulátora na reálnom zariadení. Kvôli obmedzeniam procesora na doske Arduino však museli pracovať len s lineárnymi riešeniami.

Mihai v [5] sa zaoberal tlakováním malej nádrže pomocou vzduchového kompresora riadeného regulátorom. Cieľom autora bolo vytvorenie experimentálnej platformy na riadenie tlaku vzduchu v nádobe, slúžiacej na laboratórne experimenty a akademické účely. Ako riadiaci prvok využil mikrokontrolér PIC16F877 od spoločnosti Microchip, ktorý bol parallelné zapojený spolu s priemyselným zariadením UDC 1700 od spoločnosti Honeywell. Na základe pokusných charakteristík si potom vytvoril algoritmus, ktorý v závislosti od chyby nameranej hodnoty voči nastavenej hodnote v slučke upravoval hodnotu vstupného PWM signálu. S vytvoreným riadiacim algoritmom sa mu podarilo zstrojiť funkčný systém regulácie tlaku v nádobe, ktorého reálne namerané hodnoty sa líšili od teoretických len o 2%.

Autori v [9] riešili problematiku riadenia tlaku v komore s použitím servoventilu. Na začiatku vytvorili matematický model zahrňujúci komoru, pumpu, trubicu a servomotor. Pomocou tohto modelu následne odsimulovali reguláciu tlaku pomocou PID regulácie v prostredí Simulink, kde v rámci simulácie porovnávali rozdiel medzi požadovanými a odsimulovanými hodnotami. Na základe tejto simulácie, ktorá potvrdila ich matematický model, vytvorili reálny regulátor tlaku, s využitím prostredia Labview na tvorbu riadiaceho programu. Následné experimenty dokázali nelinearitu systému spôsobenú zmenami tlaku. Problém so zistenou nelinearitou autorí vyriešili pomocou zmeny parametrov PID regulátora interpoláciou. Výsledné hodnoty ukázali, že nimi vytvorená regulácia pomocou servoventilu dosahovala rýchlu odozvu systému a vysokú presnosť požadovaných tlakových rozsahov.

V článku [17] autori opisujú spôsoby riadenia tlaku vzduchu v komore pomocou elektromagnetického mikroventilu na pneumatické riadenie. Na meranie tlaku v mikrokomore využili miniatúrny senzor tlaku Xcl-080 fungujúci na princípe ohýbania membrány. Riadenie tlaku v komore vyskúšali ovládať troma metódami riadenia

1. Bang-Bang
2. PWM
3. kompozitné riadenie

Pri metóde Bang-Bang s klesajúcou dolnou hodnotou ohraničenia klesala aj horná hodnota ohraničenia. Taktiež sa pri zmene dolnej hranice doba odozvy systému takmer vôbec nezmenila. V prípade PWM ovládania si boli doby odozvy systému s meniacou sa frekvenciou, rovnako aj s meniacou sa hodnotou chyby, podobné, trošku dlhšie ako pri použití metódy Bang-Bang. Pri použití kompozitného riadenia mal systém dobrú efektivitu a kratší čas odozvy ako pri PWM. Kolísanie tlaku a hodnota chyby boli v tomto prípade pri ustálenom stave nižšie ako v prípade použitia Bang-Bang. Systém mal plynulý priebeh sledovania hodnoty tlaku a pri zmene referencie sa dokázal pomerne rýchlo ustáliť. Celkový výsledok experimentov ukázal, že pomocou kompozitného riadenia sa v tomto prípade dokázala výrazne zlepšiť väčšina parametrov riadenia.

Shiee a kol. v článku [11] porovnali efektívnosť riadenia tlaku proporcionálneho tlakového regulátora od spoločnosti Festo a výrazne lacnejšieho a ľahšieho solenoidového ventila zap./vyp. Maximálny prietok obidvoch regulačných prvkov obmedzili pomocou regulačného ventilu prietoku. Experiment dokázal, že solenoidový ventil má kratší čas odozvy

ako tlakový regulátor, a pri zavedení sínusového vstupu má aj lepšiu sledovaciu vlastnosť. Experiment na overenie robustnosti systému vykonali náhľou zmenou objemu nádoby pri ustálenom tlaku. Zistili, že aj voči takejto zmene je solenoidový ventil odolnejší viac, ako regulátor tlaku. Na základe výsledkov zhodnotili, že solenoidový ventil je v rámci testovaných podmienok funkčnosťou dostatočný ekvivalent tlakového senzora, je však omnoho výhodnejší vďaka výrazne nižšej cene a hmotnosti.

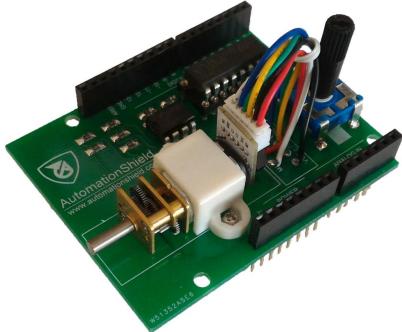
1.2 AutomationShield

Zariadenie PressureShield, ktorého tvorbou sa zaoberá táto práca, je súčasťou konceptu s názvom AutomationShield [8]. Jedná sa o voľne dostupnú (angl. open-source) iniciatívu, ktorej cieľom je vytváranie hardvérových a softvérových prostriedkov slúžiacich na výučbu automatizácie a mechatroniky. Základným prvkom je návrh a následná realizácia prototypu modulu (tzv. shield) s využitím prototypizačnej mikrokontrolorovej dosky Arduino. Na rozdiel od iných laboratórnych zariadení na výučbu a prezentáciu spätného riadenia, ktoré stoja desiatky tisíc eur, zariadenia AutomationShield sú navrhnuté tak, aby sa ich cena pohybovala v hodnotách päť eur. Vďaka tomu si každý študent, resp. vyučujúci, môže skompletovať vlastné zariadenie, na ktorom môže následne robiť pokusy. K dispozícii sú návrhy PCB dosiek, súbory potrebné na 3D tlač a taktiež kompletný zoznam potrebných komponentov, ktoré sa dajú zakúpiť. Ku každému zariadeniu je vytvorené minimálne jedno programátorské rozhranie a taktiež názorné príklady spätnoväzbového riadenia (napr. PID regulácia, LQ regulácia, MPC regulácia). Všetky rozhrania aj príklady sú zahrnuté v knižnici, ktorá je voľne dostupná. Hlavné vývojové prostredie využívané na riadenie AutomationShield modulov je Arduino IDE, kde sa využívajú C/C++ kódy, ktoré sa po stiahnutí knižnice dajú jednoducho implementovať. Na riadenie sa však využívajú aj prostredia MATLAB, Simulink, Python a iné, ktoré majú taktiež vytvorené vlastné AutomationShield knižnice. Ktoré prostredia sú použité záleží od konkrétnej dosky. Všetky knižnice a príklady použitia sa dajú stiahnuť na GitHube AutomationShieldu [14].

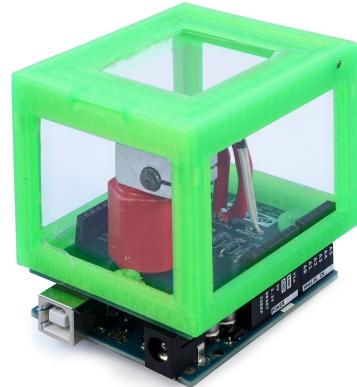
Ako príklad uvediem dva moduly. Prvý z nich, MotoShield, je zobrazený na Obr. 1.1a. Jedná sa o modul vybavený motorom s redukciou prevodov slúžiacim ako akčný člen a spätnoväzbovým encoderom s Hallovým efektom slúžiacim ako výstup. Môže slúžiť pre účely príkladu modelovania a identifikácie, riadenia rýchlosť alebo polohy [16]. Druhý modul na Obr. 1.1b sa nazýva HeatShield. Ide o modul demonštrujúci tepelnú reguláciu vykurovacieho bloku, kde akčný člen predstavuje vykurovacia kazeta a NTC odporový senzor je využívaný ako výstup s využitím jednoduchej SISO spätnoväzbovej slučky. Medzi vykurovanými časťami a PCB doskou je izolácia, ktorá zabráňuje trvalému poškodeniu dosky v dôsledku vysokej teploty, ktorá môže vystúpať na hodnotu 80 °C. Celá vykurovaná časť je chránená priehľadným bezpečnostným krytom [13].

1.3 PressureShield R1

Prvá verzia modulu PressureShield, ktorá bola vytvorená sa nazýva BlowShield. Táto verzia bola navrhnutá a vytvorená skupinou študentov Ústavu automatizácie, merania a



(a) MotoShield [16].



(b) HeatShield [13].

Obr. 1.1: AutomationShield moduly

aplikovanej informatiky, Strojníckej fakulty STU, ktorej členom som bol aj ja, na predmete s názvom Mikropočítače a mikroprocesorová technika. Hlavným cieľom bolo navrhnutie hardvéru a jeho realizácia, vytvorenie API v C/C++ a jedného príkladu spätnoväzbovej regulácie, konkrétnie s využitím PID regulátora [10].

Základným princípom tejto verzie bolo natlakovanie pretlakovej nádoby pomocou akčného člena v podobe pumpy ovláданej PWM signálom. V nádobe bol umiestnený tlakový senzor, pomocou ktorého sa získaval výstup zo sústavy, konkrétnie pretlak v nádobe oproti atmosférickému tlaku v okolí resp. oproti tlaku, ktorý bol v nádobe nameraný pred spustením pumpy pri inicializácii dosky. Na únik tlaku sa využíval prirodzený únik spôsobený netesnosťami medzi doskou a nádobou.

Návrh schémy aj PCB bol vytvorený v softvéri DipTrace. Tlakový senzor bol zapojený konektormi na 3.3 V napätie, zem, a na komunikáciu využíval protokol I2C, čo znamená, že bol zapojený na konektory SCL a SDA. Puma sa aktivovala pomocou MOSFETu, ktorého brána (angl. gate) bola zapojená na digitálny konektor. Aktivácia tohto konektora uzavrela obvod a aktivovala pumpu. Využitý bol pri tejto verzii taktiež potenciometer na manuálne určenie referencie, odpory, SMD dióda chrániacu pred spätnou elektromotorickou silou a kompenzujúci kondenzátor. Jeho umiestnenie na tomto návrhu sa ale ukázalo ako chybné, preto bolo pri kompletizácii potrebné, aby nebol osadený. Nádoba bola vytlačená pomocou 3D tlačiarne, rovnako ako držiak, v ktorom bola osadená pumpa.

Čo sa týka softvéru modulu BlowShield, bolo preň vytvorené API zahrňujúce všetky potrebné ovládacie funkcie. Taktiež bol vytvorený príklad PID regulácie, ktorá dané API využíva. Samotné API bolo vytvorené pomocou dvoch knižníc

`BlowShield.h`

`BlowShield.cpp`

Doska BlowShield sice obsahovala potenciometer, no na doske však neboli žiadny prepínač, ktorým by si mohol užívateľ vybrať, či sa mala referencia určovať ručne potenciometrom alebo sa mala určovať podľa vopred zadefinovaných hodnôt určených v kóde.

Pre pôvodnú verziu preto museli byť vytvorené dva kódy s príkladmi PID regulácie. Jeden pre automatickú referenciu, druhý pre manuálnu. Spôsob určovania referencie sa tak nastavil podľa nahratej verzie programu. Tieto príklady boli:

`BlowShield_PID_Auto.ino`
`BlowShield_PID_Manual.ino`

Celý koncept BlowShield bol navrhnutý, zrealizovaný a otestovaný. Hoci sa ukázal ako plne funkčný a príklady splnili svoj účel prezentovania spätnoväzbového riadenia, stále sa na ňom dalo veľa zlepšiť. Pri testovaní boli objavené určité chyby v návrhu PCB a stále sa dalo zahrnúť do konceptu veľa vylepšení. Taktiež bolo vytvorené len jedno API a jeden príklad regulácie, no moduly AutomationShield majú potenciál na fungovanie pomocou viacerých API s rôznymi príkladmi regulácie, nie len s príkladom PID regulátora. S týmito predpokladmi a myšlienkami som začal pracovať na vytvorení druhej verzie, ktorú som pomenoval PressureShield. V podstate sa dá povedať, že PressureShield je R2 verzia vychádzajúca z pôvodného konceptu BlowShield. A práve touto verziou R2, jej vylepšeniami hardvéru a vytvoreným novým softvérom sa zaoberá táto práca.

2 Návrh hardware

PressureShield vychádza z konceptu BlowShieldu, ktorý bol jeho predchodom. Po osadení a otestovaní pôvodnej dosky však boli objavené určité chyby, ktoré bolo treba odstrániť. Taktiež som chcel nejaké komponenty pridať. Preto som vytvoril novú schému a návrh PCB.

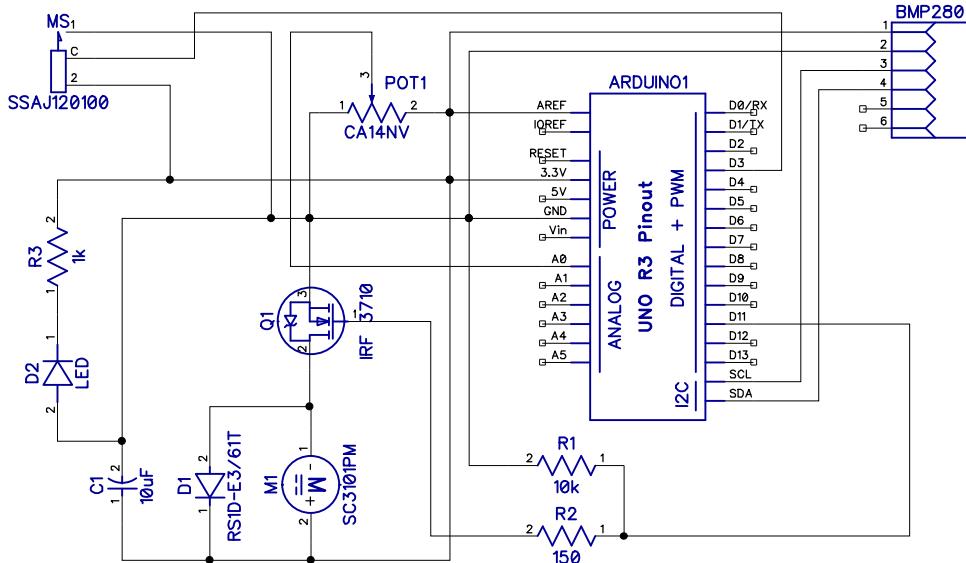


Obr. 2.1: PressureShield.

2.1 Schéma a návrh PCB

Prvým krokom k výrobe dosky plošných spojov bol návrh jej schémy. Do schémy som zakomponoval všetky prvky, ktoré bolo potrebné na doske osadiť. Na Obr. 2.2 je zobrazená schéma zapojenia dosky PressureShield. Hlavný modul zobrazený v schéme je mikrokontrolér Arduino verzia UNO R3, na ktorého konektory sa doska napája. Ďalej je zobrazené zapojenie senzora BMP280, čerpadla SC-3101PM (M1) s ochranným kondenzátorm a diódou. MOSFET zapojený v schéme na vetvu uzemnenia pumpy slúži na aktiváciu pumpy pri jeho zopnutí. Ďalej sú v schéme zahrnuté ochranné rezistory, zapojenie LED signalizujúcej napájanie v doske a odpor znižujúci napätie privedené na LED na potrebnú hodnotu. Potenciometer v schéme (POT1) slúži na nastavenie referencie systému pokial je doska pomocou spínača SSAJ120100 (MS - manual switch) prepnutá do manuálneho režimu. Taktiež je možné si všimnúť, že hoci má použitý senzor šesť konektorov, využívajú sa len štyri z nich. Prvé dva slúžia na napájanie (napätie a zem), 3. a 4. konektor slúžia na I2C komunikáciu pomocou prepojenia s konektormi SDA a SCL. Piaty a šiesty konektor nie sú zapojené. Napájanie je v schéme prepojené s konektorm AREF (analog reference).

Vďaka tomuto prepojeniu všetky analógové konektory na doske Arduino začnú fungovať na 3.3 V logike. Tento fakt je dôležitý kvôli cieľu návrhu celej dosky na 3.3 V logiku.

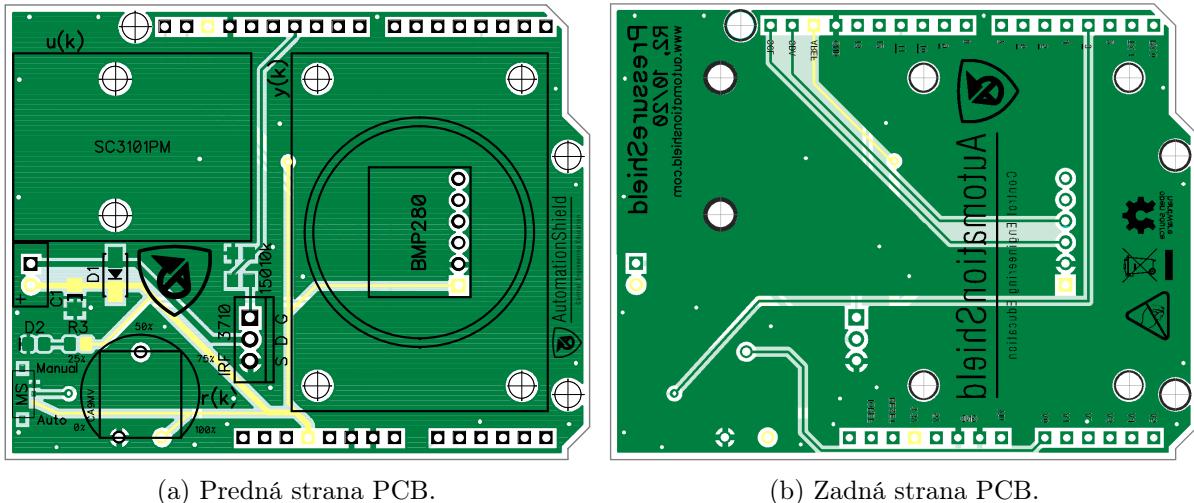


Obr. 2.2: Schéma zapojenia PressureShield.

Po navrhnutí schémy som sa mohol pustiť do návrhu samotného PCB. Na vytvorenie návrhu som použil softvér DipTrace. Každé zariadenie AutomationShield má jednotný rozmer dosky, ktorá je svojím tvarom a veľkosťou kompatibilná s doskami Arduino UNO. Kvôli obmedzeným rozmerom dosky bolo PCB vytvorené ako obojstranné, čo znamená, že horná a dolná strana dosky sú dve samostatné vrstvy obsahujúce vodivé prepojenia. Na hornej strane PCB boli osadené všetky komponenty, dolná strana slúži na spoje, ktoré sa nedali viest' hornou stranou. Prepojenie spojov medzi hornou a dolnou stranou je tvorené prekovanými otvormi. Hoci PressureShield PCB vychádza z návrhu BlowShield PCB, rozmiestnenie komponentov bolo pozmenené, taktiež bolo potrebné upraviť určité prepojenia. Na Obr. 2.3a je zobrazené rozloženie komponentov na vrchnej strane PCB. Je vidieť, kde presne sa nachádzajú plôšky, na ktoré sa majú osadiť SMD komponenty. Taktiež je zobrazené umiestnenie MOSFETu a potenciometra okolo dier, kde sa majú zaspájkovať. Okrem toho je na doske vyznačené aj to, kde presne je potrené osadiť komponenty vytlačené na 3D tlačiarni, a ktoré časti dosky tieto komponenty prekryjú. Žltou farbou je zobrazená vetva, ktorá je napojená na napájanie 3.3 V. Zadná strana PCB je zobrazená na Obr. 2.3b. Na tejto strane sa nenachádzajú žiadne plôšky na osadenie komponentov, sú tu umiestnené len vety medzi konektormi a informácie o doske.

Kvôli funkcionálite dosky boli vykonané tri zásadnejšie zmeny oproti prvotnej verzii. Prvou bola zmena umiestnenia kondenzátora C1. Pôvodne bol tento kondenzátor umiestnený medzi žilami napájania pumpy, čo sa však ukázalo ako chybné riešenie. Toto zapojenie spôsobovalo nefunkčnosť pumpy. Kvôli tomuto sa zmenilo umiestnenie kondenzátora medzi kladný pól pumpy a uzemnenie dosky. Druhá zmena bolo pridanie manuálneho posuvného prepínača. Pôvodná doska mala možnosť prepnutia medzi automatickou zmenu referencie a manuálnou zmenou pomocou potenciometra len softvérovo, kde kód nahraný v Arduine určoval, ktorý druh nastavovania referencie sa bude využívať. Aby sa mohla táto

zmena vykonávať aj bez potreby prepisovania programu v mikrokontroléry a kvôli možnosti zmeny tohto nastavenia tzv. „za jazdy“, pridal som posuvný prepínač SSAJ120100. Tento prepínač sa nenachádza v databáze DipTrace, kvôli čomu bolo potrebné vytvoriť preň umiestnenie kontaktných plôch podľa rozmerov v technickej dokumentácii [3]. Jeho stredný konektor je napojený na konektor mikrokontroléra číslo 5, zvyšné dva konektory sú napojené jeden na napätie 3.3 V a druhý na uzemnenie. Vďaka tomuto sa načítava podľa polohy prepínača na konektore číslo 5 buď logická 0 v prípade prepnutia na uzemnenie alebo logická 1, v prípade prepnutia na napätie. Prepínač som umiestnil na okraj dosky tak, aby bolo možné prepínať ho bez priestorového obmedzenia komponentov na PCB doske alebo komponentov na doske Arduino. Poslednou vykonanou zmenou bolo umiestnenie LED, ktorá signalizuje, že je doska pod napäťom, keďže doska prekrýva signálizačnú LED umiestnenú priamo na Arduine. Spolu s LED bolo potrebné na dosku pridať aj odpor, ktorý upravuje napätie, keďže pôvodné napätie 3.3 V by LED okamžite trvalo poškodilo.

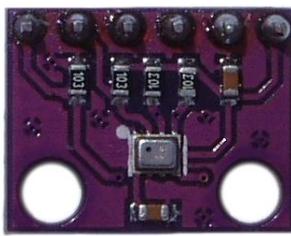


Obr. 2.3: Návrh dosky plošných spojov

2.2 Senzor tlaku

Meranie tlaku je v prípade PressureShieldu veľmi dôležitá činnosť, keďže na efektívne riadenie je potrebné mať čo najlepšiu predstavu o stave tlaku v pretlakovej nádobe. Mojim cieľom bolo použiť senzor, ktorý dokáže merať zmeny oproti atmosférickému tlaku aspoň o 1 kPa s odchýlkou maximálne ± 1 hPa. Nakoniec som sa rozhodol použiť senzor na meranie tlaku a teploty BMP280.

Tento senzor má rozsah napájania od 1.8 V do 3.6 V, rozsah merania od 300 hPa do 1200 hPa a dokáže komunikovať cez komunikačné protokoly I2C a SPI. Na mojej doske sa napája napäťom 3.3 V a na komunikáciu využíva protokol I2C.



Obr. 2.4: Senzor na meranie tlaku a teploty BMP280.

2.3 Pneumatické čerpadlo

Akčným členom mojej sústavy je pneumatické čerpadlo. Pri výbere som si musel nájsť čerpadlo, ktoré dokáže vytvoriť potrebný pretlak, aby bola možnosť vytvoriť dostatočne veľký rozsah pretlaku v komore, potrebný na reguláciu a meranie. K tomuto účelu som si vybral pneumatické čerpadlo SC-3101PM. Čerpadlo bolo prepojené s pretlakovou nádobou pomocou pneumatickej hadičky. Čerpadlo sa rovnako ako senzor napája napäťom 3.3 V, na jeho aktiváciu sa využíva MOSFET IRF 3710, ktorý je zapojený medzi čerpadlom a zemou.



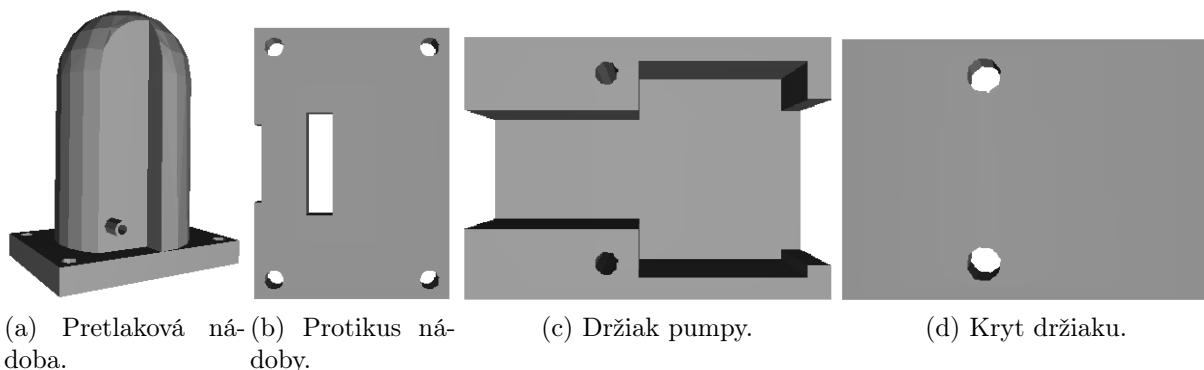
Obr. 2.5: Pneumatické čerpadlo SC-3101PM.

Pri aktivácii motora čerpadla by mohlo dôjsť k prudkému nárastu elektrického prúdu, čo by malo za následok aktiváciu prúdovej ochrany na doske Arduino v podobe vratnej poistky a vypnutie resp. resetovanie mikrokontroléra. Aby sa predišlo tomuto javu, zapojil som medzi plusový pól motora a uzemnenie dosky tantalový SMD kondenzátor s kapacitou $10 \mu\text{F}$. Tento kondenzátor zároveň slúži na kompenzáciu nárastov a poklesov prúdov v obvode. Taktiež som paralelne zapojil SMD diódu slúžiacu ako ochrana pred spätnou elektromotorickou silou.

2.4 3D komponenty

Keďže bolo potrebné skúšať rôzne verzie pretlakovej nádoby, a zároveň som mal na doske obmedzené miesto na upevnenie pretlakového čerpadla, najlepším riešením bolo vytvoriť si nádobu aj držiak čerpadla pomocou 3D tlače, pričom sa obidva tieto komponenty skladajú z dvoch častí. Samotná pretlaková nádoba (Obr. 2.6a) má tvar podobný valcu,

pričom na jednom boku má zrezanú stranu kvôli otvoru, ktorým sa privádza vzduch z čerpadla. Na spodnej strane je diera rozmerov senzora. Zvyšok spodnej časti je hrubá vrstva, v ktorej sú štyri diery na skrutky upevňujúce nádobu na plošnej doske. Medzera medzi nádobou a plošnou doskou sa izoluje pomocou obojstrannej penovej lepiacej pásky. Z druhej strany plošnej dosky sa pripievňuje medzi skrutky a dosku protikus nádoby (Obr. 2.6b) rozmermi odpovedajúcimi spodnej časti nádoby. V protikuse je medzera kvôli kontaktom senzora. Prvý dôvod upevnenia protikusu je obmedzenie priameho kontaktu kovových skrutiek o sklolaminát dosky. Druhý dôvod je rovnomernejšie rozloženie tlaku pôsobiaceho na spojené komponenty.



Obr. 2.6: Návrh 3D komponentov.

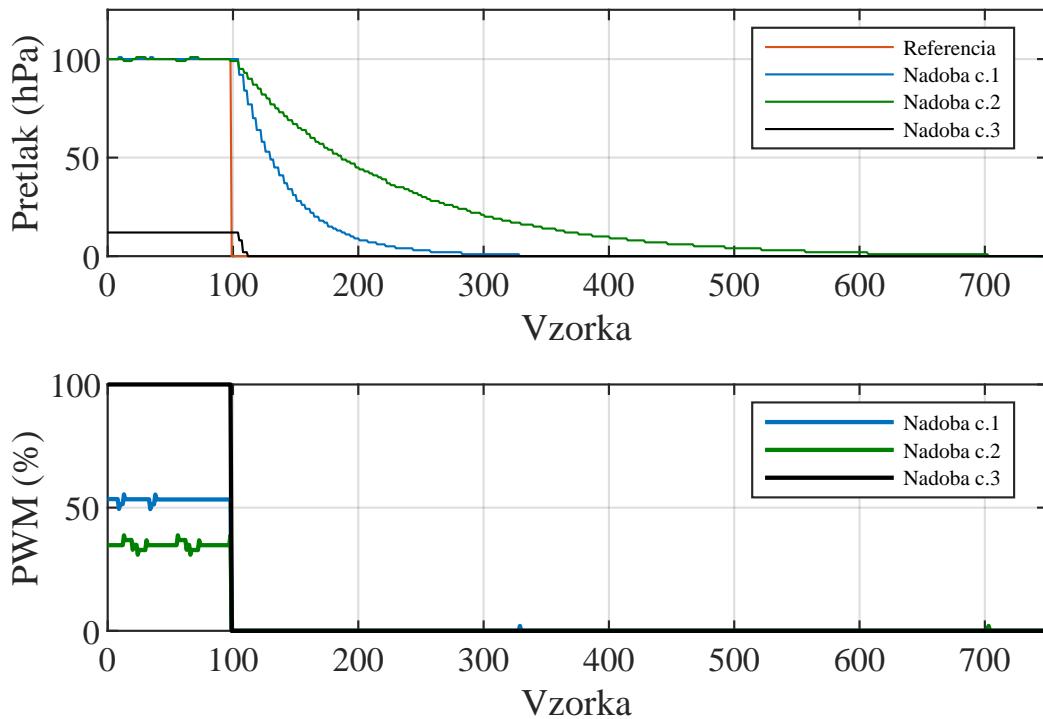
Samotný držiak čerpadla (Obr. 2.6c) sa pomocou dvoch samorezných skrutiek pripievá priamo na dosku. Po vložení čerpadla do držiaku sa pomocou dvoch ďalších dvoch samorezných skrutiek pripievá vrchná časť (Obr. 2.6d), ktorá zabráňuje vybratiu čerpadla z držiaku.

V rámci 3D tlače som narazil na problém s tlačou pretlakovej nádoby. Keďže sa na pokles tlaku v nádobe nevyužíva žiadny akčný člen, ale len prirodzený únik, hrúbka steny nádoby a jej štruktúra majú veľký vplyv na tento jav. V rámci pokusov boli dokopy vytlačené tri pokusné pretlakové nádoby, každá s inými parametrami tlače. Tieto nádoby boli postupne testované v rámci vývoja API a príkladov regulácie, aby sa zistilo, ktorá bude najviac prirodzeným únikom tlaku vyhovovať parametrom systému.

Na otestovanie úniku tlaku z každej z nádob som využil testovací program napísaný v prostredí Arduino IDE. Tento program založený na princípe PID regulácie natlakoval vnútro nádoby na pretlak 100 hPa, počkal, kým sa pretlak ustálil na tejto hodnote a následne vypol napájanie pumpy, vďaka čomu mohol pretlak plynule unikať. Pokus bol vykonaný na každej z troch nádob. Porovnanie priebehov prirodzeného úniku tlaku z nádob je zobrazené na Obr. 2.7.

Prvým jasným faktom vyplývajúcim z priebehov je to, že nádoba č.3 je určite nevhodná. Podľa všetkého sú jej steny natoľko pôrovité, že sa v nej nedokáže ani pri plnom výkone pumpy dosiahnuť väčší pretlak ako 12 hPa.

Priebehy nádob č.1 a č.2 sú podobné, obidve nádoby sa dajú natlakovať na 100 hPa s veľkou rezervou výkonu pumpy. Rozdiel medzi nádobami je ten, že nádoba č.2 tesní kvalitnejšie, takže únik z nej je pomalsší. Pri experimentálnom testovaní som však zistil, že



Obr. 2.7: Pribehy úniku tlaku v jednotlivých nádobách.

únik z nádoby č.2 je pre môj systém až príliš pomalý a spôsobuje problémy s reguláciou. V tejto nádobe sa síce dal rýchlo dosiahnuť požadovaný tlak pri zvýšení hodnoty referencie, avšak pokial sa hodnota referencie znížila, trvalo príliš dlho, kým sa nádoba odtlakovala na požadovanú hodnotu. Z tohto dôvodu som vytváral API aj všetky príklady spätnoväzbovej regulácie s použitím nádoby č.1, ktorá sa dokázala natlakovať bez problémov a únik z nej bol pri znížení referencie dostatočne rýchly na efektívnu reguláciu.

Na vytlačenie týchto troch nádob boli použité iné tlačiarne aj iný materiál (filament). Zatiaľ čo nádoby č. 2 a č. 3 boli vytlačené na jednej tlačiarni s použitím zhodného materiálu, nádoba č. 1 bola vytlačená na druhej tlačiarni s použitím iného materiálu. Z tohto dôvodu nie je možné presne určiť, či za vyhovujúce vlastnosti nádoby č. 1 môže materiál, ktorý bol použitý, alebo spôsob tlače. Približné nastavenia pri tlači jednotlivých nádob sú vypísané v Tab. 2.1.

Tabuľka 2.1: Nastavenia pri 3D tlači.

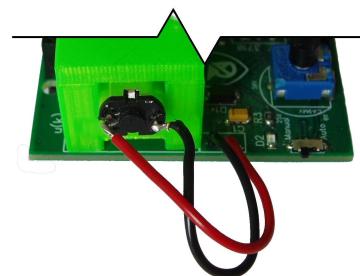
Nádoba č	Materiál	Výška Vrstvy	Typ výplne	Hustota výplne
1	PLA	0.2 mm	špirálová	20%
2	PETG	0.15 mm	Gyroid	30%
3	PETG	0.2 mm	Gyroid	25%

2.5 Montáž

Keďže má PressureShield obmedzený rozmer, niektoré súčiastky sú po skompletovaní ťažko dostupné. Z tohto dôvodu bolo potrebné dobre si naplánovať tento proces. Ako prvé bolo najlepšie zaspájkovať SMD komponenty spolu s manuálnym prepínačom. Následne som osadil senzor, pri ktorom sa sice využívajú len 4 konektory, kvôli lepšej odolnosti voči poškodeniu bolo však zaspájkovaných všetkých 6. Plôšky nevyužitých konektorov nie sú pripojené. Ďalej som osadil MOSFET a potenciometer a zaspájkoval som všetky konektory pripájajúce zriadenie na Arduino. Po namontovaní 3D komponentov bolo potrebné skrátiť jeden vývod pumpy, kvôli nedostatočnému priestoru na osadenie pri pôvodnej dĺžke (viď. Obr. 2.8a). Po osadení pumpy do držiaka som ju prepojili káblami s konektormi na PCB (viď. Obr. 2.8b). Ako poslednú som nasadil pneumatickú hadičku. Kvôli nepresnostiam pri 3D tlači nedokáže hadička dokonale utesniť priestor medzi svojou vnútornou stenou a vonkajšou stenou vývodu z nádoby. Z tohto dôvodu je potrebné potrieť vonkajšiu stenu vývodu sekundovým lepidlom a na ešte nezaschnuté lepidlo nasadiť hadičku, kvôli vytvoreniu dodatočného tesnenia. Tesnenie medzi hadičkou a vývodom pumpy je dostatočné.



(a) Zrezanie pumpy.



(b) Zapojenie pumpy.

Obr. 2.8: Úpravy pri osadení.

2.6 Zoznam komponentov a cena

Cieľom iniciatívy AutomationShield je tvorba zariadení, ktoré sú cenovo dostupné aj v prípade poskladania viacerých kusov. Túto myšlienku som sa snažil zachovať aj v prípade zariadenia PressureShield. Zoznam súčiastok, rovnako ako ich približná cena sú uvedené v Tab. 2.2.

Tabuľka 2.2: Zoznam a cena kusov potrebných na zostavenie.

Názov	Označenie a hodnota	PCB	Ks	Cena	Spolu
Pumpa	Miniatúrna vákuová pumpa: 3 V, 160 mA, 30 kPa (napr. SC3101PM)	SC3101PM	1	3.77	3.77
PCB	FR4, 2 vrstvy, 1.6 mm hrubý	-	1	2.39	2.39
Tlakový senzor	Modul tlakového senzora (napr. BMP280)	BMP280	1	2.00	2.00
Manuálny prepínač	SPDT, SMD, ON - OFF, posuvný prepínač (napr. Alps Alpine SSAJ120100)	MS	1	1.46	1.46
3D tlač	28 g, $\phi = 1.75$ mm, PETG filament, svetlo zelený, na 240 °C	-	4	1.13	4.52
N-Mosfet	IRF 3710, TO-220, (napr. Infineon IRF3710PBF)	IRF 3710	1	1.09	1.09
Dióda	SMD, 300 V, 1 A, 150 ns (napr. VISHAY RSID - E3/61T)	D1	1	0.33	0.33
Kondenzátor	10 μF , 16 VDC, SMD, 20% (napr. KEMENT T491A106M016AT)	C1	1	0.32	0.32
Tesnenie	Akrylová objestranná lepiaca páska(napr. 3M VHB GPH 110 25MMX5M)	-	1	0.20	0.20
Odpor	1 kΩ, 0805, 0.1 %, 0.125 W (napr. ROYAL OHM TC0525B0100T5)	R3	1	0.19	0.19
LED	0805, červená, (napr. OPTOSUPPLY OSR50805C1E)	D2	1	0.18	0.18
Potenciometer	10 kΩ, 250 mW, jednoočkočkový, (napr. ACP CA14NV12,5-10KA2020)	POT1	1	0.17	0.17
Odpor	150 Ω, 0805, 0.5 %, 0.125 W (napr. ROYAL OHM 0805S8F1500T5E)	150	1	0.14	0.14
Gombik potenciometra	Pre "POT1", (napr. ACP CA9MA9005)	-	1	0.12	0.12
Odpor	10 kΩ, 0805, 0.1 %, 0.125 W (napr. VISHAY CRCW080510K0FKTA)	10k	1	0.12	0.12
Skrutka	M3 × 16	-	4	0.10	0.40
Samorezná skrutka	M3 × 8	-	4	0.09	0.36
Pneumatická hadička	4 × 11 mm (napr. Festo PUN-4X0,75-SW)	-	1	0.09	0.09
Matica	M3	-	4	0.07	0.28
Červený kábel	0.5 × 70 mm	-	1	0.02	0.02
Čierny kábel	0.5 × 70 mm	-	1	0.02	0.02
Spolu:				18.10 €	

3 Programátorské rozhrania

Každý hardvér potrebuje k svojej funkčnosti spoľahlivý softvér. Aby bolo možné vytvoriť kvalitný softvér, je potrebné zvoliť si vhodné vývojové prostredie a programovací jazyk, ktorý je s daným prostredím kompatibilný. Zároveň je potrebné ovládať zvolený jazyk na dostatočnej úrovni. Pokiaľ je cieľom zariadenie programovať vo viacerých prostrediach a jazykoch, treba pre každé prostredie a programovací jazyk vytvoriť nejaké rozhranie pre programovanie aplikácií, ktoré bude slúžiť pri rôznych adaptáciách programu na zadefinovanie volaných knižníc, funkcií a parametrov. Takéto programátorské rozhranie sa nazýva API.

3.1 AutomationShield API

V koncepte AutomationShield sa zariadenia programujú v troch rôznych vývojových prostrediach, takže sa na každé zariadenie vytvárajú tri API. Následne sa ku každému API vytvorí minimálne jedna aplikácia resp. príklad použitia na demonštráciu funkčnosti zariadenia. Rovnako ako celý koncept Arduino, aj API a aplikácie spadajúce pod AutomationShield sú voľne dostupné (tzv. open-source).

Vývojové prostredia, ktoré sa v AutomatonShield využívajú, sú:

1. Arduino IDE
2. MATLAB
3. Simulink

V každom z uvedených prostredí má AutomationShield vytvorené knižnice zahrňujúce programátorské rozhrania a vytvorené príklady pre jednotlivé zariadenia. Tieto rozhrania obsahujú základné funkcie, ktoré sú potrebné pre funkčnosť zariadenia, ako je napr. funkcia na čítanie hodnôt zo senzora, funkcia využívajúca akčný člen, funkcia čítajúca referenciu z potenciometra na doske a podobne. Po stiahnutí kódu je následne na spojazdnenie zariadenia potrebné len implementovať do konkrétneho prostredia knižnicu, ktorá je preň určená. Každé z týchto prostredí sa výrazne lísi a vytvorenie jednotlivých rozhraní zahrňuje vždy problematiku špecifickú pre dané vývojové prostredie. Keďže je PressureShield súčasťou AutomationShield konceptu, vytvoril som tri rôzne programátorské rozhrania, každé v jednom z uvedených prostredí.

3.2 PressureShield rozhranie pre Arduino IDE

V tejto podkapitole presnejšie popíšem, čo vlastne je Arduino IDE, ako celé vývojové prostredie funguje a podrobnejšie sa popíše ako funguje mnou vytvorené programátorské rozhranie pre PressureShield.

3.2.1 Čo je Arduino IDE?

Arduino IDE (z angl. Integrated Development Environment) je aplikácia na princípe textového editora využívajúca programovacie jazyky C a C++, niektoré príkazy v nej sú však špecificky upravené. Slúži na napísanie, kompilovanie a nahrávanie softvéru s názvom Sketch pre vývojové dosky Arduino.

Sketch sa primárne skladá z dvoch funkcií:

- Setup - spustí sa jeden krát na začiatku pri štarte systému,
- Loop - cyklicky sa opakuje celý čas, pokiaľ systém beží.

Užívateľ si môže napisať ľubovoľné vlastné funkcie, musí ich však zavolať v jednej z dvoch hlavných funkcií. Proces komplikácie je u Arduino IDE podobný ako v iných komplátoroch s výnimkou transformačného kroku. Najskôr sa pridá knižnica Arudino.h, úprava riadkov, funkcií setup, loop a užívateľom vytvorených funkcií, a upraví sa sketch súbor na C++ súbor v programe Arduino Builder. Tento program potom skompiluje upravený sketch súbor, súbory zabudovaných knižníc (napr. pre vstup, výstup, komunikáciu atď.) a základné súbory potrebné pre komplikáciu vychádzajúce z jadra komplikácie pre Arduino. Súbor v jadre definujúci hlavnú funkciu pri komplikácii má názov main.cpp. Záver komplikácie spočíva z prepojenia týchto súborov a ich transformácie do binárneho kódu, ktorý sa môže nahrať do mikroprocesora. Arduino Builder taktiež zálohуje výsledne súbory aby tak urýchlił ďalšie komplikácie. Celá aplikácia Arduino IDE je vytvorená autormi celého konceptu Arduino, takže rovnako ako všetky súčasti konceptu je voľne dostupná (open-source) [6].

3.2.2 PressureShield API

Rovnako ako iné programovacie jazyky, aj C++ využívané v Arduino IDE podporuje vytváranie knižníc. Prvým krokom k tomu, aby niekto mohol používať ktorékoľvek zariadenie AutomationShield v rámci prostredia Arduino IDE, je stiahnutie si celej knižnice AutomationShield z portálu GitHub a jej nainštalovanie v tomto prostredí. V mojom prípade bolo potrebné, rovnako ako vo všetkých projektoch AutomationShield, vytvoriť knižnice zahrňujúce ovládacie funkcie môjho zariadenia, aby som ich mohol jednoduchým načítaním využiť vo viacerých riadiacich programoch a nemusel ich zbytočne duplikovať zvlášť pre každý súbor. Pre potreby tohto projektu som si vytvoril dve základné knižnice:

PressureShield.h

PressureShield.cpp

Knižnica PressureShield.h je tzv. hlavičkový súbor (angl. header file). Nachádza sa v nej definícia základných funkcií a ich vstupno/výstupné parametre, zadefinovanie

registrov načítavaných z tlakového senzora, zahrnutie zdrojových knižníc, pomenovanie a zadefinovanie používaných konštant a podobne. Knižnica PressureShield.cpp je zdrojový súbor (angl. source file), v ktorom je zahrnutá predošlá knižnica. V tejto knižnici sú už napísané kódy jednotlivých riadiacich funkcií, ktoré sú potrebné pre chod zariadenia. Kód programátorského prostredia pre PressureShield je uvedený v dodatku A, konkrétnie PressureShield.h sa nachádza v dodatku A.1, PressureShield.cpp v dodatku A.2.

Aby sa vytvorené knižnice mohli využívať, je potrebné zahrnúť ich do sketchu. Na zahrnutie obidvoch knižníc stačí privolať len jednu knižnicu pomocou príkazu:

```
#include <PressureShield.h>
```

Treba pripomenúť, že aby tento príkaz fungoval, užívateľ musí mať vo svojom programe Arduino IDE nainštalovanú knižnicu AutomationShield. AutomationShield je založený na objektovo orientovanom programovaní (OOP, z angl. Object-oriented programming), čo znamená, že moje zariadenie som zadefinoval ako objektovú štruktúru PressureShield, ktorá spadá pod triedu (angl. class) s názvom PressureClass. Pod touto triedou sú v hla-vičkovom súbore zadefinované všetky funkcie, premenné a konštanty, ktoré PressureShield využíva. Na konci súboru je zadefinovaný objekt PressureShield, ktorý sa následne využíva v sketchoch Arduino IDE. Po zadefinovaní objektu a zahrnutí knižníc, sa môžu v rámci hlavného programu začať využívať funkcie. Aj keď sa trieda v ktorej sú zahrnuté volá PressureClass, v programe sa volá objekt. Výsledný príkaz, ktorým sa funkcia privolá je tak v tvare

```
PressureShield.nazovFunkcie();
```

Prvá funkcia, ktorá je volaná v Setup časti sketchu je funkcia begin. Pokiaľ je na začiatku sketchu zahrnutú knižnicu PressureShield.h, zavolá sa príkazom

```
PressureShield.begin();
```

V tejto funkcií sa nachádzajú základné nastavenia dosky. Najskôr sa pre každý z použitých konektorov zadefinuje, či sa jedná o vstupný alebo výstupný konektor. Dalej sa nastaví konektor, ktorým sa aktivuje pumpa na hodnotu 0, aby sa predišlo náhodnému spusteniu pumpy počas inicializácie. Keďže je doska navrhnutá na 3.3 V logiku, a na náš konektor definujúci analógovú referenciu (AREF) ide priamo 3.3 V napätie, nastavíme vo funkcií príkazom analogReference(EXTERNAL) referenciu analógových konektorov na toto napätie. Ďalej sa inicializuje I2C komunikáciu medzi senzorom a doskou Arduino a načítajú sa funkcie, ktoré merajú tlak na senzore. Posledným príkazom v tejto funkcií je 5 sekundové oneskorenie (delay), aby sa v prípade, že tesne pred inicializáciou bola spustená pumpa, stihol vyrovnať tlak v pretlakovej nádobe s okolitým tlakom.

Po zadefinovaní základných nastavení dosky, je spojazdnená I2C komunikácia a je možné čítať hodnoty namerané senzorom. Najskôr je však potrebné senzor kalibrovať. Keďže moje zariadenie slúži čisto na meranie pretlaku v nádobe oproti atmosférickému tlaku, najlepším riešením je zadefinovať si hodnotu atmosférického tlaku ako referenciu a merat len zmeny oproti nej. Keďže je atmosférický tlak premenlivý, nemôže sa dať ako relatívna nula v celom API konšstanta. Tento problém rieši funkcia

```
PressureShield.calibration();
```

Táto funkcia hned po uplynutí 5 sekúnd, slúžiacich na vyrovnanie tlaku z predchádzajúcej funkcie, odmeria aktuálny tlak, ktorý je v nádobe. Tento tlak potom priradí

premennej `_minPressure`, ktorá sa od tohto momentu berie ako relatívna 0 v systéme. Funkcia taktiež priradzuje hodnotu 105 000 premennej `_maxPressure`, po ktorej dosiahnutí sa deaktivuje pumpa aj v prípade, že by hodnota bola pod požadovanou referenciu. Tento príkaz funguje ako ochrana senzoru, aby nedošlo k jeho poškodeniu vytvorením príliš veľkého pretlaku v nádobe. Hodnota bola určená podľa údajov, ktoré boli k dispozícii v dokumentácii senzora [4].

Aby sa mohol regulovať pretlak v nádobe, musí byť určená nejaká referencia, t.j. žiadana hodnota, na ktorej sa má tlak udržiavať. Na PressureShielde sa dá táto hodnota určiť dvoma spôsobmi. Pokiaľ je prepínač MS1 na polohe Auto, ako referencia sa berú hodnoty priamo zapísané v programe. Pokiaľ sa však prepínač prepne do polohy Manual, ako referencia sa začne brať hodnota nastavená pomocou potenciometra. Na čítanie tejto hodnoty bola vytvorená funkcia

```
PressureShield.referenceRead();
```

Prvý príkaz vo funkciu načíta hodnotu z preddefinovaného konektora PRESSURE_RPIN (analógový konektor A0 na doske Arduino) a umiestní ju do premennej s názvom `val`. Načítaná hodnota sa však musí upraviť, keďže rozmedzie analógových vstupov je v hodnotách 0 –1023 a celá referencia je obmedzená na hodnoty 0 –100. Na túto úpravu sa využíva funkcia `map`. Táto funkcia funguje na princípe zadania nameranej hodnoty, možného rozsahu tejto hodnoty a zadania žiadaneho rozsahu. V tomto prípade teda príkaz vyzerá takto:

```
map((float)val, 0.0, 1023.0, 0.0, 100.0);
```

Už upravená hodnota sa zapíše do premennej s názvom `percento`, ktorá je výstupom z tejto funkcie.

Každý regulovaný systém potrebuje nejaký akčný člen, ktorý sa aktivuje pokiaľ je meraná hodnota nižšia ako referencia a deaktivuje pokiaľ meraná hodnota referenciu prekročí. Keby sa však akčný člen iba prepínali medzi hodnotami 0 a 1, kde 0 by znamenala úplne vypnutie a 1 plný výkon, celá regulácia by bola veľmi nepresná. Kvôli tomu sa využíva PWM signál, vďaka ktorému sa dokáže v prípade potreby upraviť výkon akčného člena. PWM signál sice preskakuje len medzi hodnotami 0 a 1, ale podľa hodnoty jeho vstupu sa upravuje pomer, kedy je aktívny a kedy neaktívny, čím omnoho efektívnejšie a rýchlejšie upravuje výsledný aktívny čas. V tomto prípade je akčným členom pretlaková pumpa. Na riešenie riadenia jej výkonu, z dôvodov opísaných vyššie, sa využíva funkcia

```
PressureShield.actuatorWrite();
```

Vstupom do funkcie je akčný zásah `u`, ktorý sa vyráta v rámci programu podľa typu regulátora. Tento akčný zásah sa však musí prerátať na hodnotu zodpovedajúcu možným hodnotám PWM signálu. PWM signál môže mať hodnotu v rozsahu 0 –255, preto sa v ďalšom príkaze funkcie vynásobí akčný zásah hodnotou 255 a následne vydelí hodnotou 100, aby sa určila správna percentuálna hodnota, ktorá zodpovedá prepočtu akčného zásahu do hodnôt PWM signálu. Nakoniec sa táto hodnota zapíše na preddefinovaný konektor s názvom PRESSURE_UPIN (digitálny konektor 11 na doske Arduino), čo spôsobí potrebné zvýšenie, resp. zníženie výkonu pumpy.

Regulovaný systém, do ktorého vstupuje akčný člen, musí mať aj výstup, ktorým sa sleduje hodnota veličiny, ktorú je potrebné uregulovať, aby sa vedelo, v akom stave sa celý systém nachádza. V tomto prípade je touto sledovanou veličinou tlak, resp. pretlak, ktorého hodnota sa meria pomocou senzora tlaku a teploty BMP280 schovaného v pretlakovej nádobe. Funkcia, ktorá sa využíva na meranie tlaku je funkcia

```
PressureShield.sensorRead();
```

Mojim úmyslom ale bolo, aby som hodnotu získanú senzorom s I2C komunikáciou ne-načítaval pomocou knižníc dostupných na internete, ale pomocou vlastnej knižnice. Z toho dôvodu sa nachádza v tejto funkcií ďalšia funkcia PressureShield.readPressure(), ktorá slúži na tento účel.

Aby funkcia readPressure fungovala, bolo potrebné vytvoriť niekoľko podfunkcií. Celý proces začína funkciou `begin_config`. Táto funkcia inicializuje senzor, komunikáciu na ňom a nastavuje potrebné parametre. Podľa dokumentácie k senzoru [4] bolo zistené, že na inicializáciu senzora a jeho nastavenie na mnou požadované hodnoty je potrebné na register 0xF4 zapísat hexadecimálnu hodnotu 0b111111 a na register 0xF5 zapísat hodnotu 0b0. Po inicializácii sa prejde ku druhej funkcií s názvom `measure_config`. Táto funkcia prepíše hodnotu na registry 0xF4 na hodnotu 0b1111 a hodnotu na registry 0xF5 na hodnotu 0b1000, čím začne senzor proces merania tlaku. Po spustení merania sa začnú načítavať dátá, ktorími senzor reprezentuje tlak. Prvým krokom k načítaniu dát je funkcia `readRegister`. V nej sa zadefiniuje I2C adresa senzora. Vstupom do nej je následne konkrétny register, výstupom je hodnota načítaná z regisitra. Pomocou tejto funkcie sa následne vo funkcii `readCoefficients` načítajú hodnoty zo všetkých regisetrov, potrebné pre výpočet aktuálneho tlaku, a priradia sa príslušným premenným. Tieto hodnoty sa však už počas merania nezvyknú meniť. Hodnoty regisetrov, v ktorých sa odzrkadľuje zmena tlaku sa načítavajú vo funkciách `press_read` a `temp_read`. Kedže na prepočet tlaku je potrebná aktuálna teplota, predposledným krokom celého procesu zistenia tlaku je funkcia `readTemperature`, v ktorej sú vzorce z dokumentácie senzora [4]. Po dosadení potrebných hodnôt načítaných z príslušných regisetrov funkcia vráti hodnotu aktuálnej teploty vo vnútri nádoby. V poslednom kroku sa vo funkcii `readPressure` do vzorcov, ktoré boli taktiež získané z dokumentácie senzora, dosadia hodnoty príslušných regisetrov a aktuálnej teploty, vďaka čomu sa vypočíta aktuálna hodnota tlaku.

Výstupom funkcie `readPressure` je hodnota aktuálneho tlaku vstupujúca do funkcie `sensorRead`. Okrem získania hodnoty aktuálneho tlaku sa v tejto funkcií nachádza tak tiež už vyššie spomínaná poistka, ktorá vypne pumpu pokial' sa pretlak v nádobe dostane k hodnote, ktorá by mohla poškodiť senzor BMP280. Posledným príkazom v tejto funkcií sa upravuje nameraná hodnota podľa premenných `_minPressure` a `_maxPressure` určených pri kalibrácii, aby výstup z funkcie zodpovedal pretlaku v požadovanom rozsahu.

V rámci vytvorenia príkladu LQ regulátora, je potrebné mať funkciu, vďaka ktorej je možné odhadnúť vnútorný stav systému pomocou rovníc Kalmanovho filtra. Preto sa v hlavičkovom súbore `PressureShield.h` zahrnula do knižnice funkcia `getKalmanEstimate.inl`, ktorá je súčasťou `AutomationShield` súborov a slúži presne na tento účel. Funkcia sa môže následne v rámci programu použiť v tvare

```
PressureShield.getKalmanEstimate(X, u, y, A, B, C, Q_Kalman, R_Kalman);
```

kde u je hodnota vstupu systému, y je hodnota výstupu systému, \mathbf{A} , \mathbf{B} , \mathbf{C} , \mathbf{Q} , \mathbf{R} sú matice vychádzajúce z identifikovaného matematického modelu a X je výstup z funkcie. Aby sa mohli tieto matice zadať do rovnice, musí sa zahrnúť do knižnice taktiež knižnica `BasicLinearAlgebra.h`, vďaka ktorej sa dokážu zapísat matice v potrebnom tvare. Takýto zápis je potom v tvare `BLA::Matrix<riadky, stĺpce>`. Ako príklad sa môže uviesť zápis matice \mathbf{A} , ktorý vyzerá takto:

```
BLA::Matrix<3, 3> A = {0.993, 0.01526, -0.003223, 0.4206};
```

3.3 PressureShield rozhranie pre MATLAB

3.3.1 Čo je MATLAB?

Programovacia platforma MATLAB sa využíva na výpočty, modelovanie, simulácie, programovanie a vytváranie aplikácií. Na rozdiel od rozhrania Arduino IDE nie je voľne dostupná. MATLAB využíva svoj špecifický maticovo orientovaný programovací jazyk. Umožňuje vyriešenie technických problémov založených na maticovej alebo vektorovej formulácii v krátkom čase v porovnaní s potrebou napísat kód riešiaci rovnakú problematiku napr. v jazyku C. MATLAB umožňuje vytvorenie dvoch rôznych programových súborov, skript alebo funkciu. V skriptových súboroch môže užívateľ zapísat sériu príkazov, ktoré sa majú vykonať spoločne. Skript neprijíma vstupy ani nevracia výstupy, pracuje čisto s dátami v pracovnom priestore (angl. workspace). Súbory funkcií dokážu prijímať vstupy a vraciať výstupy, pričom interné premenné sú pre funkciu lokálne [2]. Pre aplikáciu špecifických riešení existujú ponuka MATLAB toolboxy, ktoré umožňujú užívateľovi oboznámiť sa s určitou špecializovanou problematikou a technológiou a zároveň vyriešiť v danej problematike konkrétné úlohy. Toolboxy sú komplexné zbierky funkcií MATLABu, ktoré rozširujú toto programové prostredie o riešenia konkrétnych problémov napr. spracovanie signálu, programovanie mikrokontroléra Arduino, neurónových sietí atď.

3.3.2 PressureShield API

Aby sa mohlo využívať vývojové prostredie MATLAB na nahrávanie programu do dosky Arduino, je potrebné stiahnuť si na to príslušné rozšírenie s potrebnými ovládačmi. To sa dá vykonať pomocou ikony Add-Ons, kde je potrebné vyhľadať výraz „Arduino“. Po zobrazení výsledkov treba nájsť a nainštalovať balík MATLAB Support Package for Arduino Hardware. Hoci sa vytvorili všetky potrebné knižnice a funkcie na riadenie PressureShieldu v jazyku C++ v rámci vývojového prostredia Arduino IDE, žiadna z nich nie je kompatibilná s prostredím MATLAB. Kvôli tomuto je potrebné vytvoriť ich znova v jazyku prostredia MATLAB. Prvým krokom, rovnako ako v prípade API pre C++, je nainštalovanie knižnice AutomationShield do MATLABu, aby mal prístup ku všetkým potrebným súborom, ktoré sa budú využívať pri tvorbe ďalších funkcií. Táto inštalácia sa vykoná pomocou súboru, ktorý je obsiahnutý v knižnici s názvom

```
installForMATLAB.m
```

po otvorení ktorého stačí, aby užívateľ spustil daný script (tlačidlo Run). V prípade úspešnej inštalácie knižnice vypíše príkazový riadok (angl. Command Window) hlásenie: „AutomationShield MATLAB API added to MATLAB path“. Po tejto inštalácii som sa mohol pustiť do tvorby vlastných súborov. Prvým mnou vytvoreným súborom je súbor

`PressureShield.m`

ktorý vytvára MATLABovskú triedu s názvom `PressureShield`. V tejto triede sú definované všetky základné premenné a funkcie, ktoré sú potrebné pre chod zariadenia `PressureShield` v prípade jeho ovládania cez prostredie MATLAB.

Prvou vytvorenou funkciou je inicializačná funkcia s názvom `begin`. Úlohou tejto funkcie je identifikovať pripojenú dosku Arduino a priradiť ju k príslušnej premennej, taktiež definuje I2C komunikáciu so senzorom a inicializuje nastavenia senzora. K účelu identifikácie dosky Arduino sa v rámci vytvorenej triedy `PressureShield` vytvorila objektová štruktúra s názvom `PressureShieldObject`, pod ktorú budú spadať všetky využívané premenné v triede. Na identifikáciu dosiek Arduino existuje v MATLABe funkcia `arduino()`, ktorá spresní cez aký port je doska pripojená, o ktorý typ dosky sa presne jedná a aké konektory sú na doske k dispozícii. Samotný príkaz v mojej funkcií vyzerá takto:

```
PressureShieldObject.arduino = arduino();
```

Tento príkaz vytvorí prepojenie medzi doskou a MATLABom a pridelí všetky zistené informácie o doske pod premennú `arduino` v rámci celého objektu. Vďaka tomu nie je potrebné vytvárať prepojenie a definíciu dosky pri ďalších funkciách, stačí sa v nich odvolať na už existujúce prepojenie pomocou objektu `PressureShieldObject.arduino`. Po úspešnej identifikácii dosky vypíše príkazový riadok hlásenie: „`PressureShield initialized`“.

Aby sa mohla zadefinovať I2C komunikácia so senzorom, je potrebné najskôr vedieť, na akej I2C adrese sa tento senzor nachádza. Táto adresa sa dá zistiť pomocou MATLAB funkcie `scanI2CBus`, ktorej výstupom je adresa, na ktorej je dostupná komunikácia, v tomto prípade komunikácia so senzorom. Keď bola zistená adresa senzora, bolo potrebné vytvoriť premennú, v ktorej by bol senzor pomocou jeho adresy zadefinovaný. K tomuto účelu som využil MATLAB funkciu `device`, ktorá dokáže vytvoriť potrebnú komunikáciu. Výsledný príkaz, slúžiaci k tomuto účelu je

```
dev = device(PressureShieldObject.arduino, 'I2CAddress', '0x76');
```

kde:

- `PressureShieldObject.arduino` je použité zariadenie Arduino v rámci objektovej štruktúry,
- `dev` je premenná, do ktorej sa komunikácia na danej adrese zadefinovala pre ďalšie použitie,
- `I2CAddress` definuje, že zariadenie komunikuje pomocou I2C protokolu,
- `0x76` je konkrétna adresa použitého senzora na zbernici.

Senzor BMP280 je potrebné pred jeho použitím inicializovať. Tento krok sa vykonáva mnou vytvorenou funkciou s názvom `BMP280Init`, ktorá sa volá v rámci funkcie `begin`. Vstupom do funkcie je zadefinované I2C zariadenie pod názvom `dev`. Táto funkcia využíva príkazy `writeRegister`, pomocou ktorých zapíše na inicializačné registre hexadecimálnu hodnotu, potrebnú pre spustenie merania s požadovanými parametrami (register F4 - hodnota 0b1111, register F5 - hodnota 0b1000). Názorne taký príkaz vyzerá:

```
writeRegister(dev, 'F5', 0b1000);
```

Z dôvodu potreby mať možnosť zistiť hodnotu na potenciometri, v prípade manuálneho nastavenia referencie, sa aj v tomto prípade vytvorila funkcia, ktorá je schopná túto hodnotu načítať. Podobne ako v C++ knižnici, funkcia sa volá `referenceRead`. Na načítanie hodnoty využíva MATLAB funkciu `readVoltage`, ktorá je schopná načítať hodnotu z analógového konektora dosky A0, na ktorý je výstup potenciometra napojený. Výsledný zápis funkcie je:

```
reference = readVoltage(PressureShieldObject.arduino, PressureShieldObject.  
Pressure_RPIN) * 100 / 5
```

`Pressure_RPIN` je preddefinovaný konektor A0 v rámci celého objektu. Celkové napätie sa následne vynásobí hodnotou 100 predstavujúcou 100% rozsahu a vydelí hodnotou 5, ktorá predstavuje maximálne možné napätie na konektore. Aj keď je potenciometer napájaný len 3.3 V napäťim, vďaka upraveniu referencie analógových konektorov na toto napätie je rozsah stále v pôvodných hodnotách 0 – 5. Pomocou týchto prepočtov sa výsledná referencia na potenciometri vyjadrí ako zodpovedajúca hodnota vstupného napäťia v rozmedzí 0 – 100.

Rovnako ako v prípade funkcií vytvorených v jazyku C++, aj v MATLABe sú dvomi najdôležitejšími funkciami funkcie pre riadenie akčného člena, čiže v tomto prípade riadenie napájania pumpy, a funkcia, ktorá dokáže načítať dátu z tlakového senzora a následne pomocou výpočtov určiť aktuálnu hodnotu tlaku v nádobe. Prvá funkcia riadiaca akčný člen sa v mojej `PressureShield` triede pre MATLAB volá rovnako ako v C++ `actuatorWrite`. V celej tejto funkcií sa nachádza len jedna MATLABovská funkcia s názvom `writePWMDutyCycle`, ktorá slúži na generovanie PWM signálu na potrebnom digitálnom konektore dosky. Celá funkcia aj s potrebnými parametrami je v tvare

```
writePWMDutyCycle(PressureShieldObject.arduino, PressureShieldObject.  
Pressure_UPIN, (percent / 100));
```

kde `PressureShieldObject.arduino` je zadefinované spojenie s doskou vytvorené v predchádzajúcej funkcií, ďalší parameter `PressureShield.Pressure_UPIN` je preddefinované označenie pre digitálny konektor číslo 11 na doske Arduino, pomocou ktorého sa ovláda napájanie pretlakovej pumpy a `percent` je vstup do funkcie, ktorý zodpovedá potrebnej úrovni napájania pumpy vyrábanej v riadiacom programe podľa stavu referencie a aktuálneho tlaku.

Funkcia `sensorRead` slúži na načítanie aktuálnej hodnoty tlaku v nádobe. Rovnako ako v prípade knižnice v C++, aj v MATLABe sa k tomuto účelu vytvorila podfunkcia, v tomto prípade s názvom `BMP280`.

Vstupom do podfunkcie BMP280 je premenná so zadefinovaným senzorom dev. Táto funkcia využíva na načítanie hodnôt z registrov senzora funkciu `readRegister`. Faktom však je, že hodnoty na väčšine týchto registrov sa počas merania nemenia, sú to len akési vzorkovacie a konverzné hodnoty. Taktiež teplota, ktorej hodnota sa využíva na prepočet tlaku, sa počas pomerne krátkej doby používania senzora nemení v takom rozsahu, aby to malo významný vplyv na hodnotu tlaku. Kvôli čo najlepšej optimalizácii som sa rozhodol, že hodnoty, ktoré sa nemenia, rovnako ako teplotu, stačí načítať iba raz a potom pracovať s rovnakými údajmi. Z tohto dôvodu sa pri prvom spustení funkcie BMP280 vykoná cyklus, ktorý sa pri ďalších spusteniach už nevykonáva. Tento cyklus najskôr načíta hodnoty z registrov slúžiacich na výpočet teploty a taktiež z registrov, ktorých hodnota sa počas merania nemení. Následne sa v cykle pomocou hodnôt z registrov vypočíta, vzorcami získanými z dokumentácie senzora [4], hodnota teploty. Keďže sa tieto hodnoty získavajú len raz, no pre funkciu senzora sú potrebné počas každého využitia funkcie, ukladajú sa do premenných typu `persistent`, vďaka čomu sa ich hodnota zapamätá. Druhá časť funkcie sa opakuje vždy pri spustení. Hodnota meniaca sa so zmenou tlaku sa nachádza na troch registroch, ktoré treba spojiť dokopy. Aby sa vytiahli potrebné bity z jednotlivých registrov, využíva sa príkaz `bitget`, v ktorom je zakaždým zadefinované, ktoré konkrétnie bity sa majú do pomocnej premennej načítať. Pre získanie potrebej hodnoty sa nakoniec bity zo všetkých troch registrov spoja a premenia sa do decimálneho tvaru. Po získaní všetkých potrebných údajov sa tieto hodnoty dosadia do vzorcov, ktoré pochádzajú z dokumentácie senzora, a na výstupe funkcie sa zobrazí aktuálny tlak v nádobe.

Príklad načítania bitov z registrov, ich spojenia do jedného bitu a konverzie na decimálny tvar:

```
UP_MSB = readRegister(dev,'F7','uint8');
UP_LSB = readRegister(dev,'F8','uint8');
UP_XLSB = readRegister(dev,'F9','uint8');

UP_MSBbit = bitget(UP_MSB,8:-1:1);
UP_LSBbit = bitget(UP_LSB,8:-1:1);
UP_XLSBbit = bitget(UP_XLSB,8:-1:5);
bitsP = [UP_MSBbit UP_LSBbit UP_XLSBbit];
adc_p = bi2de(bitsP,'left-msb');
```

Výsledný tlak, ktorý je výstupom podfunkcie BMP280 je zároveň hodnota, ktorá sa načíta po zavolení pôvodnej funkcie `sensorRead`.

3.4 PressureShield rozhranie pre Simulink

3.4.1 Čo je Simulink?

Simulink je doplnkový produkt vývojového prostredia MATLAB. Poskytuje interaktívne grafické prostredie na simuláciu, modelovanie a analýzu dynamických systémov pomocou rýchlej výstavby virtuálnych prototypov. Modelovanie v Simulinku sa uskutočňuje pomocou grafického užívateľského rozhrania GUI. Celý model sa vytvára pomocou schémy, ktorá sa uloží v tvare názov.slx. Sú v ňom k dispozícii preddefinované bloky v rámci zahrnutých knižníč, ktoré sa dajú do schémy modelu jednoducho pridať pomocou akcie chyť a pusť (angl. drag-and-drop) alebo dvojklikom na plochu schémy a napísaním názvu bloku,

ktorý chceme použiť. Dajú sa v ňom využívať lineárne aj nelineárne systémy v spojiteľnom aj diskrétnom čase. Vďaka možnosti meniť jednoducho parametre sa dajú s ľahkosťou odsimulovať rôzne varianty modelu. Keďže je Simulink v podstate len rozširujúcou časťou MATLABu, je možné bez problémov zdieľať dátu z jedného prostredia do druhého.

3.4.2 PressureShield API

Aj keď je Simulink súčasťou MATLABu, rozšírenie slúžiace na ovládanie Arduina pre MATLAB nie je použiteľné aj v rámci Simulinku. Z tohto dôvodu je potrebné stiahnuť a nainštalovať ďalší rozširujúci modul. Inštalácia prebieha rovnakým postupom ako pri MATLABe, kliknutím na ikonu Add-Ons sa zobrazí prehliadač, v ktorom treba vyhľadať výraz „Arduino“ a z vyhľadaných výsledkov nainštalovať rozšírenie Simulink Support Package for Arduino Hardware. Rovnako ako v dvoch predchádzajúcich prostrediacich, aj pre Simulink existuje vytvorená AutomationShield knižnica, ktorú je potrebné nainštalovať, aby sa mohol Simulink používať na programovanie a ovládanie zariadení tohto konceptu. Táto inštalácia sa vykonáva cez prostredie MATLAB pomocou skriptu vytvoreného na tento účel. V AutomationShield knižnici sa nachádza súbor

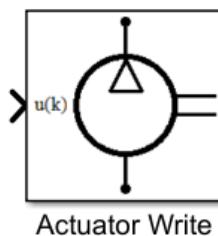
`installForSimulink.m`

po ktorého spustení, pokiaľ je inštalácia úspešná, vypíše príkazový riadok hlásenie: „AutomationShield Simulink API added to MATLAB path“. Každé z AutomationShield zariadení má v tomto Simulink API vytvorenú knižnicu, ktorej súčasťou sú bloky obsahujúce ovládacie funkcie.

PressureShield ma taktiež vytvorenú svoju knižnicu, v ktorej sa nachádzajú základné riadiace bloky a funkcie. Jej názov je

`PressureLibrary.slx`

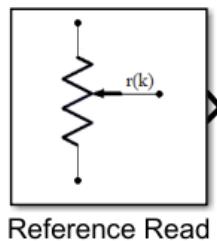
Prvým z ovládacích blokov knižnice je blok s názvom Actuator Write. Vstupom do tohto bloku je hodnota akčného člena u , ktorá sa ďalej pomocou MATLAB funkcie `ActuatorWrite Function` prepočíta na hodnotu zodpovedajúcu PWM signálu v rozmedzí 0 – 255. Výstup z tejto funkcie sa nakoniec pomocou Simulink bloku PWM zapíše na konektor ovládajúci pretlakovú pumpu. Blok je zobrazený na Obr. 3.1.



Obr. 3.1: Simulink blok Actuator Write.

Rovnako ako v predošlých prípadoch, aj v prípade regulácie tlaku v nádobe riadením cez programy vytvorené v Simulinku, je požiadavka byť schopný určovať referenciu dvoma spôsobmi, dopredu zadefinovanými hodnotami alebo pomocou nastavenia potenciometra na doske. Kvôli tomu musel byť aj v Simulinku vytvorený blok, ktorý načítava hodnotu na potenciometri, s názvom Reference Read zobrazený na Obr. 3.2. Vstup do

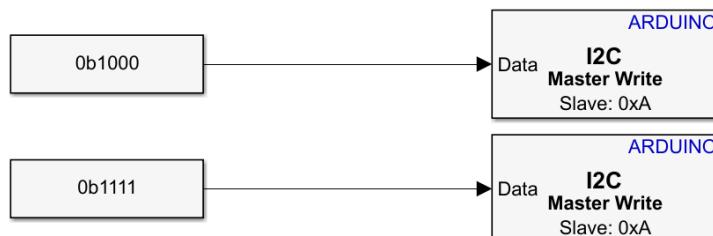
tohto bloku je hodnota analógového signálu načítaná pomocou Simulink bloku Analog Input z konektora, na ktorý je potenciometer napojený. Táto hodnota je následne pomocou bloku Data Type Conversion upravená na dátový typ Double. Takto upravená hodnota vstupuje do funkcie Reference Read, kde sa upravuje z rozmedzia 0 – 1023, teda rozmedzia hodnôt, ktoré môžu byť načítané na analógovom konektore, na hodnoty v rozmedzí 0 – 100 vyhovujúce požiadavkám zariadenia. Výstupom z bloku je upravená hodnota v tomto rozsahu.



Obr. 3.2: Simulink blok Reference Read.

Blok načítavajúci hodnotu tlaku zo senzora sa volá Sensor Read. Využíva preddefinované Simulink bloky na ovládanie Arduina v kombinácii s funkciou na načítanie tlaku vytvorenou pre potreby MATLABu. Nachádzajú sa v ňom dva podsystémy, jeden sa vykonáva iba raz, druhý sa cyklicky opakuje.

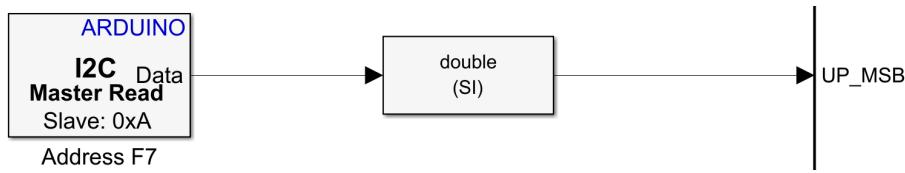
Prvý podsystém funguje rovnako, ako časť funkcie v MATLAB API, ktorá sa opakuje iba raz. Najskôr sa zinicializuje senzor pomocou hexadecimálnej konštanty vstupujúcej do bloku I2C Write, v ktorom je nastavený register, ktorého hodnotu treba prepísat. Rovnako ako v predošlých prípadoch, aj tu bolo potrebné kvôli inicializácii prepísat dva registre. Príklad tohto zapísania hodnôt je zobrazený na Obr. 3.3.



Obr. 3.3: Prepisovanie registrov v Simulinku.

Po inicializácii senzora začne funkcia načítavať údaje z registrov, ktorých hodnoty sa nemenia a z registrov potrebných na výpočet teploty, keďže aj tu sa odmeria teplota len raz. Ďalej sa pracuje s jednou konštantnou hodnotou teploty. V Simulinku sa na načítanie údajov z registra využíva blok I2C Read. Príklad načítania dát z registra v Simulinku je zobrazený na Obr. 3.4.

Načítané údaje sa potom blokmi Data Type Conversion upravia na dátový typ double, a v takomto tvare vstupujú do MATLAB funkcie s názvom BMP280Init. Táto funkcia je v podstate len modifikácia funkcie využitej v MATLAB API. Konvertuje načítané údaje do decimálneho tvaru a vypočíta teplotu prostredia. Výstupom z funkcie sú nemeniace sa konštanty vyplývajúce z načítaných registrov a teplota prostredia. Tieto

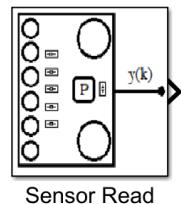


Obr. 3.4: Načítavanie hodnôt registrov v Simulinku.

údaje sú zároveň aj výstupom z celého podsstému a vstupujú do druhého podsstému, ktorý sa cyklicky opakuje. V ňom sa najskôr načítajú hodnoty z troch registrov, ktorých údaje predstavujú hodnotu tlaku v nádobe. Tieto údaje sú taktiež konvertované na dátový typ `double`, a spolu s konštantami z prvého podsstému vstupujú do MATLAB funkcie `BMP280`, kde sa využijú na výpočet aktuálneho tlaku pomocou dosadenia do vzorcov. Výstupom z funkcie je hodnota tlaku, a riadiaca premenná `i`.

Riadiaca premenná `i` slúži na zablokovanie prvého podsstému. Pri inicializácii má hodnotu 1. Jej hodnota vstupuje do MATLAB funkcie, ktorá v prípade zmeny jej hodnoty z 1 na iné číslo pomocou bloku `If` zablokuje prvý podsstém, ktorý je typu `If Action Subsystem`, čo znamená, že pokial riadiaca hodnota vstupujúca do podsstému neodpovedá nastaveným parametrom (v našom prípade sa `i` musí rovnať 1), blok sa nevykonáva. Zmena tejto premennej nastane po dvoch sekundách od spustenia programu.

Výstupom z celého bloku `Sensor Read`, je aktuálna hodnota tlaku v nádobe. Výsledný blok je zobrazený na Obr. 3.5.



Obr. 3.5: Simulink blok Sensor Read.

Po vytvorení troch základných ovládacích blokov, je potrebný ešte funkčný blok, v ktorom sa upraví ich funkcia pre potreby samotných regulátorov. Tento blok sa nazýva rovnako ako moje zariadenie `PressureShield` a je zobrazený na Obr. 3.6. Prvým vstupom do funkčného bloku je hodnota akčného člena u vyrátaná regulačným obvodom, ktorá sa zapisuje do predtým vytvoreného bloku `Actuator Write`. Druhým vstupom do funkčného bloku je hodnota tlaku načítana blokom `Sensor Read`. Táto hodnota v rámci bloku vstupuje do MATLAB funkcie `PressureShield Function`.

```

function Pressure = PressureShield(SensorRead)

persistent ref;

if isempty(ref)
ref = double(SensorRead);
end

Pressure = double((SensorRead-ref)/100);

```

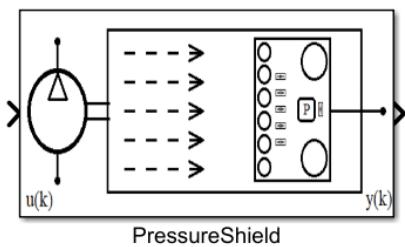
```

if (ref == 0)
ref = Pressure * 100;
end

end

```

Pri prvom spustení funkcie sa načíta hodnota tlaku v nádobe pri spustení zariadenia. Keďže sa celá regulácia zameriava na regulovanie pretlaku v nádobe, je potrebná referenčná hodnota na vypočítanie pretlaku. Táto hodnota sa zapíše do premennej `ref`. Pri ďalších meraniach sa potom referenčná hodnota odčíta od aktuálneho tlaku, vďaka čomu je výstupom z funkcie hodnota pretlaku oproti atmosférickému tlaku okolo zariadenia. Táto hodnota sa ešte vydelí konštantou 100, aby výstupný pretlak neboli v jednotkách Pa, ale v jednotkách hPa. Výsledná hodnota pretlaku je následne výstupom aj z celého funkčného bloku `PressureShield`.



Obr. 3.6: Simulink blok PressureShield.

4 Príklady spätnoväzbového riadenia

Vytvorené programátorského rozhrania pre každé vývojové prostredie umožňujú pomocou funkcií, ktoré sú v nich zahrnuté pomerne jednoducho ovládať základné funkcie zariadenia PressureShield. Aby však bolo zrejmé, ako presne sa používajú, a zároveň aby sa mohla demonštrovať ich funkčnosť, bolo potrebné vytvorenie programov s ich využitím. V mojom prípade som využil API najskôr na vytvorenie programu, vďaka ktorému som bol schopný identifikovať celý riadený systém. Ďalej som pomocou nich vytvoril rôzne príklady spätnoväzbovej regulácie pre každé vývojové prostredie, v ktorom som mal k dispozícii vytvorené API. Konkrétnie sa jedná o príklady spätnoväzbovej regulácie pomocou

1. PID riadenia
2. LQ riadenia
3. MPC riadenia

Dôvodom tvorby troch odlišných príkladov regulácie je fakt, že každé riadenie funguje na inom princípe. Vďaka tomu som mohol pomocou výstupov rôznych riadení porovnať ich funkčnosť a presnosť riadenia. Hoci sa jedná o riadenie rovnakého zariadenia resp. rovnakého systému pomocou rovnakého API, funkčnosť regulátorov je odlišná. Zatiaľ čo PID regulátor sa nastavuje pomocou dopredu zadefinovaných konštánt riadenia, ku funkčnosti LQ a MPC regulátora je potrebný matematický model.

4.1 Identifikácia a modelovanie

Ako už bolo spomenuté vyššie, na funkčnosť LQ regulátora bol potrebný matematický model. Tento model prezentuje systém v podobe matematického zápisu, konkrétnie v podobe matíc, ktoré sa využívajú na opis správania sa dynamiky systému, ktorý je potrebné uriať. Sú známe tri druhy identifikácie

1. Black box - jedná sa o experimentálnu identifikáciu, využívajúcú na formulovanie modelu informácie získané počas experimentovania s ním. Model opisuje vstupno-výstupné správanie objektu, no neumožňuje pohľad do vnútornej štruktúry [12].
2. Grey box - analyticko-experimentálna identifikácia, kde sa matematicko-fyzikálou analýzou získaný model porovnáva s experimentálnymi dátami reálneho objektu s dátami získanými numerickou simuláciou [12].

3. White box - analytická identifikácia, pri ktorej sa na základe matematicko-fyzikálnej analýzy zostavuje model, t.j. opisuje javy prebiehajúce v reálnom objekte a vzťahy medzi veličinami pomocou základných fyzikálnych princípov [12].

V prípade môjho systému som sa rozhodol využiť Black box identifikáciu. Podľa definície napísanej vyššie je však jasné, že aby som mohli využiť tento druh identifikácie, boli potrebné dátá reprezentujúce môj systém, v tomto prípade zariadenie PressureShield, ktoré sa získali na základe experimentu.

Môj experiment bol založený na využití tzv. APRBS (z angl. amplitude modulated pseudo random signal) signálu. Jednalo sa o pseudonáhodný binárny signál s danými amplitúdovými levelmi resp. ohraničeniami. Na vygenerovanie tohto signálu som využil script v MATLABe s názvom

`PressureShield(APRBS.m)`

v ktorom som si zadefinoval dĺžku signálu a jeho minimálnu a maximálnu hodnotu. V mojom prípade bolo rozhranie 0 – 5000 Pa. Tento script využíva funkciu knižnice AutomationShield s názvom `aprbsGenerate`, ktorá samotný signál vygeneruje a uloží ho v hlavičkovom súbore `aprbsU.h`, aby sa mohol ďalej využívať, v tomto prípade na identifikáciu v jazyku C++.

Na samotnú identifikáciu som si vytvoril program vo vývojovom prostredí Arduino IDE s názvom

`Identification(APRBS.ino)`

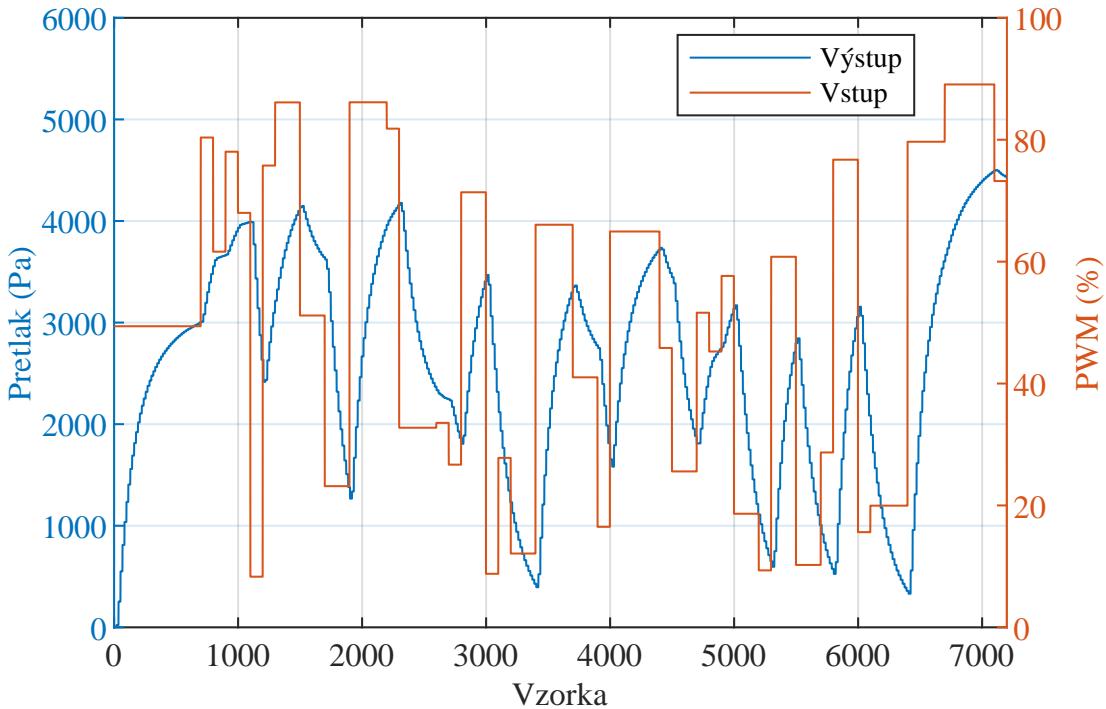
V tomto programe som najskôr zahrnul ovládacie knižnice PressureShield vytvorené v rámci API a taktiež knižnicu `aprbsU.h`, v ktorej je vygenerovaný APRBS signál. Po inicializácii a kalibráciu zariadenia sa zapisuje ako akčný člen hodnota vygenerovaného APRBS signálu, bez akejkoľvek regulácie. Výstupom z programu je neregulovaná hodnota tlaku v nádobe, ktorá sa mení čisto podľa zmeny hodnoty akčného člena. Na zaznamenanie celého priebehu signálu som využil MATLAB script s názvom

`PressureShield_Experiment.m`

ktorý využíva funkciu `readExperiment` na zaznamenanie priebehu vstupnej a výstupnej veličiny. Výsledný graf identifikačného experimentu je zobrazený na Obr. 4.1.

Po získaní experimentálnych dát, ktoré mi pomohli identifikovať moju sústavu som potreboval vytvoriť model, ktorý by mi ju matematicky opisoval, a ktorý by som mohol ďalej využívať v mojich príkladoch spätnoväzbovej regulácie. Na vytvorenie tohto modelu som využil identifikačný toolbox v prostredí MATLAB.

Najskôr som si načítal dátá získané pri identifikácii, a zadefinoval si konkrétnie hodnoty ako hodnoty vstupu a hodnoty výstupu. Taktiež som si určil vzorkovací čas. Moje dátá a vzorkovací čas som si uložil pomocou funkcie `iddata`, v tvare potrebnom na identifikáciu a modelovanie, do premennej `data`. Na aktiváciu identifikačného toolboxu je potrebné zadať v príkazovom riadku MATLABu príkaz `ident`. Po aktivácii toolboxu som si vybral ako vstupné dátá `Data object`, a ako vstup som dal moju premennú



Obr. 4.1: Priebeh vstupu a výstupu identifikácie v čase.

data. Po importovaní dát som si vybral možnosť modelovania State Space Models, v rámci ktorej som nastavil Specify value rovné dvom a Estimation Method na možnosť Regularized Reduction. Táto možnosť bola vybraná na základe viacerých pokusov modelovania, keďže model vytvorený touto metódou odhadu pôsobil ako najlepší. Výsledný model s názvom `ss1` som následne importoval do MATLAB Workspace, a pomocou príkazu `c2d` zdiskretizoval. Výsledný zdiskreditovaný model som si uložil do súboru `BlackBoxModel.mat`. Tvar príkazu, využitého na diskretizáciu modelu je:

```
BlackBoxModel = c2d(ss1,Ts,'zoh');
```

Výsledné matice, ktoré reprezentujú mnou vytvorený model, sú:

$$\mathbf{A} = \begin{pmatrix} 0.993 & 0.01526 \\ -0.003223 & 0.4206 \end{pmatrix}, \mathbf{B} = \begin{pmatrix} 0.008751 \\ -0.1501 \end{pmatrix}, \mathbf{C} = (1.74 \quad 0.8755).$$

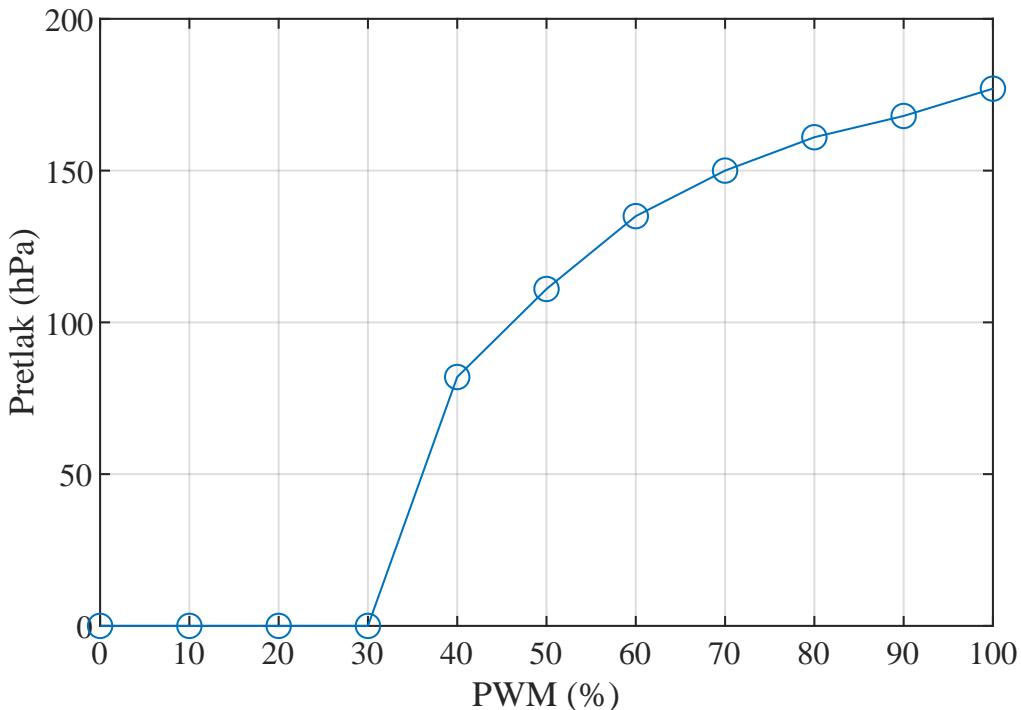
Vďaka týmto maticiam som vedel ďalej využiť môj model na čiastočnú predikciu správania sa systému, keďže pomocou nich bolo možné opísť dynamiku celého objektu.

Po úspešnej identifikácii modelu a vytvorení zdiskretizovaného modelu som sa mohol pustiť do vytvárania príkladov spätnoväzbovej regulácie.

4.2 Nelinearita systému

V prípade PressureShieldu sa na znižovanie tlaku v nádobe nevyužíva žiadny riadiaci prvok, ako napríklad ventil, ktorého pootvorenie by ovplyvňovalo únik tlaku. V mojom

prípade sa na únik tlaku z nádoby využíva iba pôrovitosť jej stien spolu s netesnosťou medzi nádobou a doskou, resp. akýsi prirodzený únik. Tento fakt má za dôsledok veľkú nelinearitu vytvoreného systému, keďže čím väčší je pretlak v nádobe, tým rýchlejšie zároveň uniká tlak. Aby som mohol demonštrovať tento jav, vytvoril som tri pokusné experimenty. Dva na zobrazenie nonlinearity, jeden na jej matematické overenie. S ich pomocou som sa pokúsil určitým spôsobom nelinearitu dokázať a čiastočne aj zdefinovať.

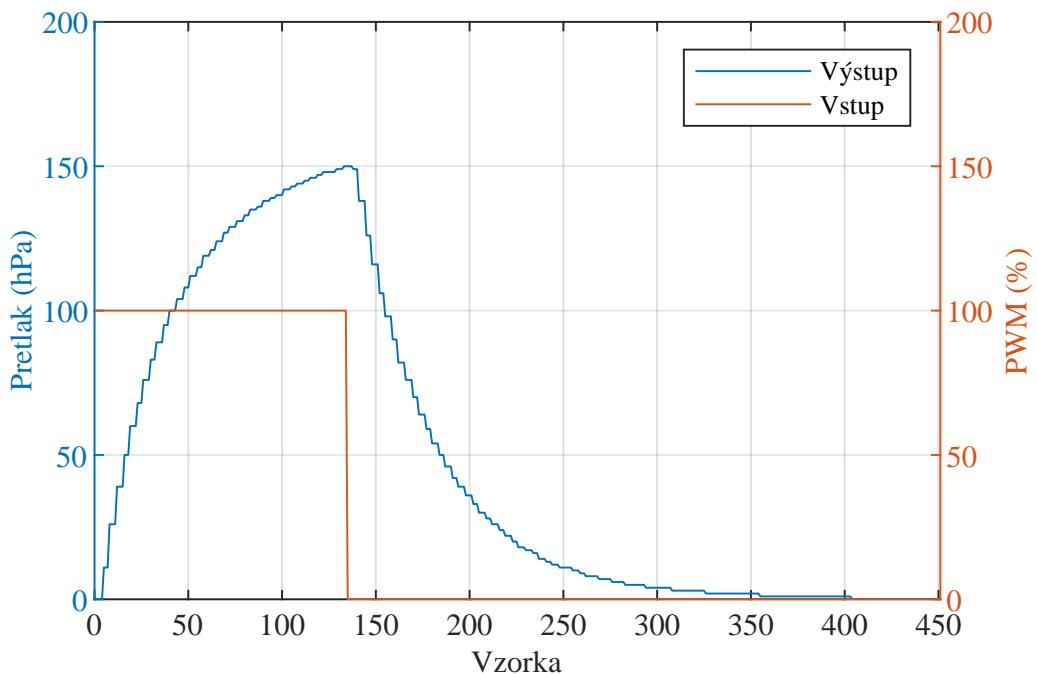


Obr. 4.2: Ustálené hodnoty tlaku pri stabilnej hodnote PWM.

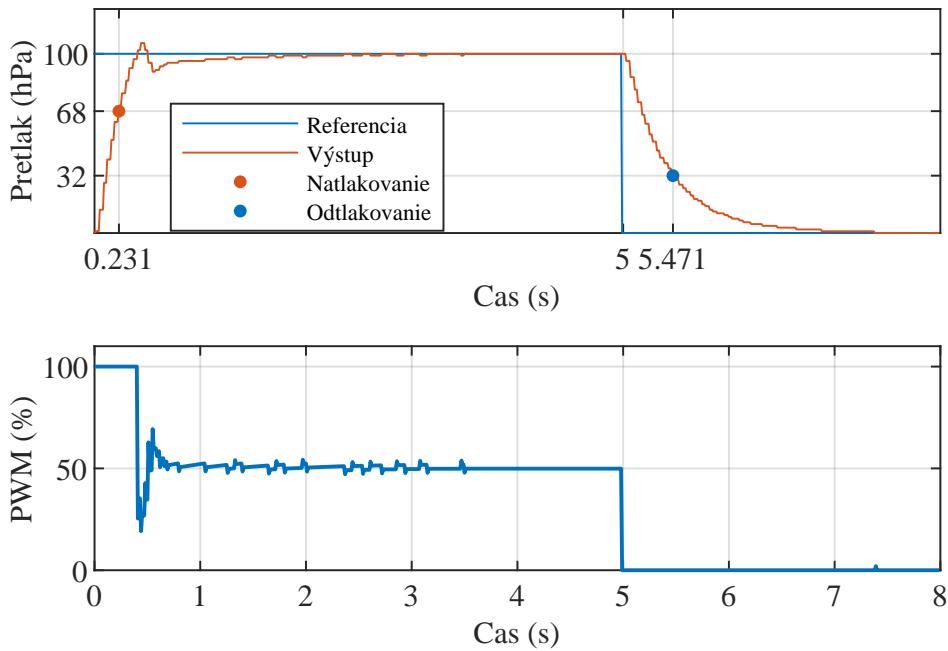
Prvý pokus, ktorý som vykonal je zobrazený na Obr. 4.2. Tento pokus spočíval v tom, že sa postupne zvyšovala hodnota PWM signálu o 10 % v celom rozsahu 0 –100 %. Pri každej hodnote PWM sa počkalo, kým sa ustálil pretlak v nádobe, ktorý je schopný udržať sa pri danom výkone pumpy. Vo výslednom priebehu je zobrazených 11 bodov, ktoré zodpovedajú ustáleným hodnotám tlaku pre konkrétné hodnoty signálu PWM. Krivka, ktorá sa vytvorila spojením týchto bodov je očividne nelineárna, čo zodpovedalo mojej prvej úvahе, že celý systém je kvôli neriadenému úniku tlaku nelineárny.

Druhý pokus, ktorý som za účelom dokázania nonlinearity systému vytvoril, spočíval v natlakovaní nádoby na určitú hodnotu a následne na jej prirodzenom odtlakovaní. Priebeh tohto pokusu je zobrazený na Obr. 4.3. Na priebehu je zjavne vidieť, že krivka natlakovania sa čiastočne líši od krivky odtlakovania nádoby. Toto zároveň z časti dokazuje, že proces natlakovania nádoby a proces odtlakovania nádoby sú v tomto prípade dva odlišné procesy.

Ako dôkaz odlišnosti týchto procesov som vykonal tretí pokus. Najskôr som odmeral čas, za ktorý sa dokáže nádoba natlakovať pri plnom výkone pumpy z 0 hPa na 68 hPa, čo predstavuje rozsah 0 –68 %. Čas natlakovania na 68 hPa bol rovný 231 ms (červený



Obr. 4.3: Priebeh natlakovania a odtlakovania nádoby.



Obr. 4.4: Časové konštanty tlaku.

bod na Obr. 4.4). Druhá časť pokusu spočívala v odtlakovaní pri vypnutej pumpe zo 100 hPa na 32 hPa, čo zodpovedalo pôvodnému rozsahu 0 – 68 % z prvej časti pokusu, v tomto prípade som si však 100 hPa určili ako 0 % na osi y a 0 hPa ako 100 % na osi y. Čas odtlakovania na 32 hPa bol rovný 471 ms (modrý bod na Obr. 4.4). Z porovnania týchto

dvoch časov možno vidieť, že proces natlakovania a odtlakovania nádoby sú rozdielne, keďže časová konšanta obidvoch procesov na rovnakom rozsahu tlaku je výrazne odlišná.

4.3 PID riadenie

Prvým typom spätnoväzbovej regulácie, ktorý som využil na vytvorenie príkladov je proporcionalno-integračno-derivačné (skrátene PID) riadenie. Jedná sa o riadenie veľmi často používané v praxi, keďže má jednoduchú matematickú štruktúru a je ľahko implementovateľné. Taktiež nie je pri jeho použití potrebné matematické vyjadrenie systému v podobe modelu. Na regulovanie využíva len výstup systému, referenciu a proporcionalny koeficient zosilnenia, integračný a derivačný koeficient času. Nevýhodou PID regulátora je, že sa dá používať iba na SISO (z angl. single-input single-output) systémy. Pokial by sme chceli pomocou PID regulátora regulovať MIMO (z angl. multiple-input multiple-output) systém, museli by sme ho najskôr rozdeliť na viac SISO systémov. Matematický výpočet vstupu pri PID regulácii je

$$u_{(t)} = K_p \left(e_{(t)} + \frac{1}{T_i} \int_0^t e_t dt + T_d \frac{de_{(t)}}{dt} \right), \quad (4.1)$$

kde:

- $u_{(t)}$ je vypočítaný vstup systému v čase t ,
- $e_{(t)}$ je odchýlka medzi žiadoucou referenciou a výstupom systému v čase t ,
- K_p je proporcionalna konšanta zosilnenia,
- T_i je integračná konšanta času,
- T_d je derivačná konšanta času.

Z rovnice 4.1 možno vidieť, že výpočet vstupu prebieha pomocou odchýlky $e_{(t)}$, ktorá sa vypočíta pri spätej väzbe odčítaním výstupu systému od požadovanej referencie. Regulátor vezme aktuálnu hodnotu odchýlky, jej integráciu v čase t a deriváciu v čase t , a následne vyhodnotí vplyv každej zložky na základe prislúchajúcej konštanty. Nevýhodou PID regulátora je možnosť vyrátania vstupu mimo potrebný rozsah, dokonca môže vyrátať záporný vstup. Aby sa predišlo problémom s takouto nežiaducou hodnotou, musí sa ošetriť úprava hodnoty vstupu mimo rozsah softvérovo.

Rovnica 4.1 je rovnica PID regulácie v spojitom čase. Aby sme však vedeli využívať náš PID regulátor pri iteračných príkladoch, potrebujeme ju pretransformovať do diskrétneho tvaru. Diskrétny tvar PID zobrazuje rovnica 4.2.

$$u_{(k)} = K_p \left(e_{(k)} + \frac{1}{T_i} \sum_{j=1}^k e_{(j)} + T_d \frac{e_{(k)} - e_{(k-1)}}{T_s} \right). \quad (4.2)$$

4.3.1 PID riadenie pre Arduino IDE

Vôbec prvým príkladom, resp. programom, ktorý som na demonštrovanie funkčnosti zariadenia PressureShield vytvoril, bol PID regulátor vo vývojovom prostredí Arduino IDE s názvom PressureShield_PID.ino. Tento príklad využíva funkcie zahrnuté v rámci mnou vytvoreného API. Prvým krokom je zahrnutie knižníc, nastavenie konštánt ovládacích parametrov ako je čas vzorkovania T_s a nastavenie referencie potrebnej v prípade automatickej regulácie. Taktiež sa na začiatku nastavujú konštandy regulátora K_p , T_i a T_d . V tomto prípade som pokusmi typu pokus a omyl prišiel na to, že regulátor sa najlepšie správa pri konštantách $K_p = 3$, $T_i = 1$ a $T_d = 0.01$.

```
#include <PressureShield.h>      // Include the library
#include <Sampling.h>            // Include sampling

unsigned long Ts = 10;           // Sampling in milliseconds
float R[] = {80.0, 50.0, 70.0, 40.0, 90.0, 30.0, 60.0, 20.0, 40.0, 70.0,
             60.0};

#define Kp 3                      // PID Kp
#define Ti 1                       // PID Ti
#define Td 0.01                    // PID Td
```

Po zadefinovaní potrebných parametrov sa v rámci inicializačnej funkcie `Setup` zadefinuje rýchlosť komunikácie so sériovým portom, vykoná sa funkcia `begin` a funkcia `calibration`, ktoré inicializujú zariadenie spolu so senzorom tlaku. Taktiež sa konštandy regulátora spolu so vzorkovaním T_s vložia do objektu knižnice AutomationShield s názvom `PIDAbs`. V rámci tohto objektu sa vypočítava hodnota akčného člena u počas chodu programu.

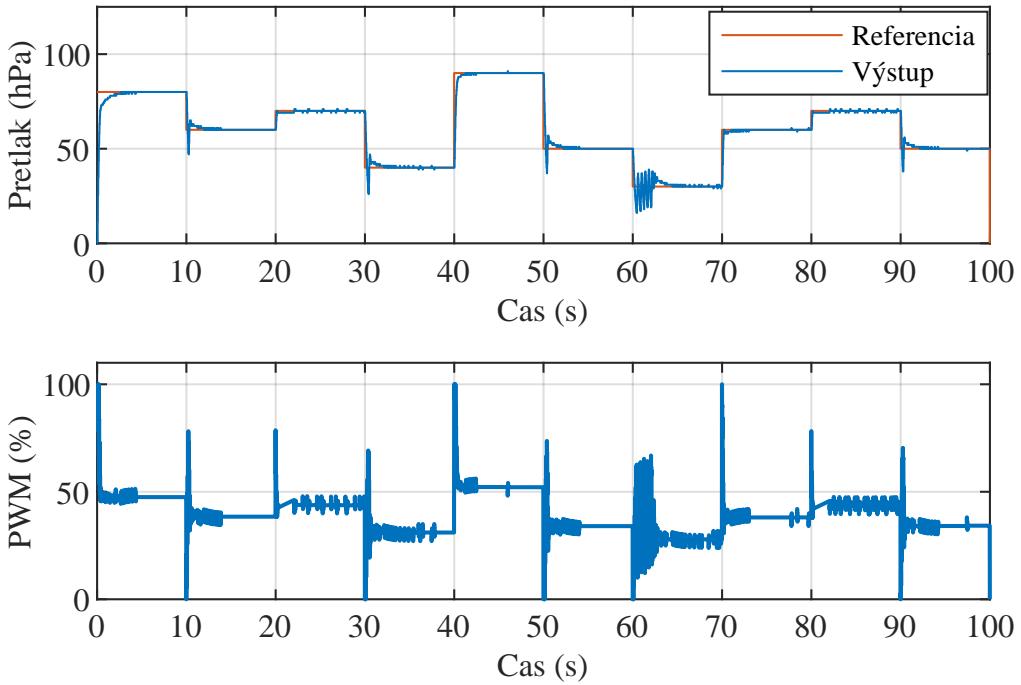
Cyklicky sa opakujúci krok programu s názvom `step` zahŕňa samotnú reguláciu. Najskôr skontroluje hodnotu nastavenú na konektore označenom MS, ktorá určuje, či sa referencia regulácie nastavuje automaticky podľa dopredu zadefinovaných hodnôt, alebo sa nastavuje ručne pomocou potenciometra na doske. Po určení referencie (v prípade manuálneho módu sa načítava pomocou funkcie `referenceRead`) načíta funkciou `sensorRead` hodnotu pretlaku v nádobe, a pomocou tejto hodnoty a určenej referencie následne objekt `PIDAbs` vypočíta hodnotu akčného člena, ktorého hodnota sa zapíše do funkcie `actuatorWrite`. Nižšie je uvedený príklad týchto operácií v prípade manuálneho režimu.

```
r = PressureShield.referenceRead();        // Read reference
y = PressureShield.sensorRead();            // Read Sensor
u = PIDAbs.compute(r-y,0,100,0,100);       // PID
PressureShield.actuatorWrite(u);            // Actuate
```

Na záver program vypíše postupne referenciu, hodnotu nameranú senzorom aj hodnotu vypočítaného akčného člena.

Ako bolo spomenuté v podkapitole 4.2, v pretlakovej nádobe dochádza k výraznému vplyvu nelinearity systému. Z tohto dôvodu nie je možné PID regulátor nastaviť tak, aby reguloval všetky hodnoty referencie rovnako kvalitne.

Ako možno vidieť na Obr. 4.5, dôsledok nelinearity je pomerne výrazný. Pri mnou nastavených hodnotách sa regulátor najlepšie správa pri hodnotách pretlaku vyšších ako 40 hPa, pri nižších hodnotách referencie sice reguluje pomerne kvalitne, avšak nie tak



Obr. 4.5: Priebeh vstupu a výstupu PID regulátora v Arduino IDE.

plynulo. Pri tomto nelineárnom systéme som však počas pokusov neprišiel na žiadne riešenie, ktorým by sa tohto javu dalo zbaviť.

Na Obr. 4.5 je zobrazený aj dôsledok nelinearity na hodnotu akčného člena. Pri hodnotách referencie nad 40 hPa sa hodnota akčného člena drží približne na rovnakej úrovni. Keď sa referencia zníži pod 50 hPa, nelinearita spôsobí, že akčný člen kolíše.

Celý kód príkladu PID regulátora vytvoreného v prostredí Arduino IDE je uvedený v dodatku A.3.

4.3.2 PID riadenie pre MATLAB

Po úspešnom vytvorení prvého príkladu PID riadenia, som sa mohol pustiť do príkladu PID riadenia v MATLABe s názvom `PressureShield_PID.m`. Rovnako ako v prípade prostredia Arduino IDE, aj tu som už mal vytvorené API, so základnými ovládačmi funkciami, ktoré tvorbu programu zjednodušilo. Prvým príkazom, ktorý sa využíva je funkcia `startScript`. Táto funkcia sa využíva namiesto obvyklých inicializačných príkazov `clear all`, `close all` a `clc`. Rozdiel medzi nimi a touto funkciou je ten, že nevymaže všetky premenné, resp. že zachová premenné potrebné na CI (z angl. Continues Integration). Potom sa vytvorí objekt `PressureShield`, ktorý sa tvorí pomocou triedy s rovnakým názvom zahrnutej v API. Rovnakým spôsobom sa vytvára objekt s názvom `PID`, ktorý je však vytvorený pomocou triedy zahrnutej v knižnici `AutomationShield`. Tento objekt sa využíva na nastavenie potrebných parametrov PID regulátora a na výpočet akčného člena u. Ďalej sa pomocou funkcie `begin` inicializuje celé zariadenie, nastavia sa regulačné konštanty K_p , T_i a T_d , nastaví sa vzorkovanie a referencia pre prípad nastavenia zariadenia na automatický mód.

```

PressureShield = PressureShield;      % Construct object
PressureShield.begin();              % Initialize shield

PID = PID;

```

Rovnako ako vo všetkých mnou vytvorených príkladoch spätnoväzbovej regulácie, aj v tomto prípade sa hodnoty referencie vzťahujú na pretlak v nádobe oproti atmosférického tlaku v okolí. Aby sa dal tento pretlak určiť, musí sa na začiatku regulácie odmerať atmosférický tlak. Následne treba túto hodnotu odčítavať od aktuálnej hodnoty v nádobe, vďaka čomu je celkový výstup pretlak v nádobe. V prípade príkladov v C++ je celá táto operácia vykonávaná v rámci API, to znamená, že výstupom z funkcie `sensorRead` je už hodnota pretlaku. V prípade príkladov v MATLABe som sa rozhodol zvoliť iný postup. Na začiatku PID regulácie sa odmeria tlak v nádobe zodpovedajúci atmosférického tlaku, resp. tlak, ktorý je v nádobe pokial ešte nebola aktivovaná pumpa. Táto hodnota sa uloží do premennej `initPressure`. Ďalej sa táto premenná v programe využíva pri upravení hodnoty tlaku na pretlak. Taktiež sa v objekte PID nastavia zadefinované regulačné konštanty pomocou príkazu `setParameters`

```

init = PressureShield.sensorRead(); % Read reference
PID.setParameters(Kp, Ti, Td, Ts); % Feed the constants to PID object

```

Po zadefinovaní potrebných parametrov sa začne v programe pomocou funkcie `while` vykonávať regulačná slučka. Najskôr sa odmeria tlak v nádobe. Od tejto hodnoty sa potom odčíta premenná `initPressure` a výsledok sa vydelí číslom 100, aby bola výsledná hodnota v hPa. Druhým krokom je určenie referencie, po ktorom sa pomocou funkcie objektu PID s názvom `compute` vypočíta požadovaná hodnota akčného člena. Táto hodnota sa v ďalšom kroku zapíše do funkcie `actuatorWrite`, ktorá aktivuje pumpu s potrebným výkonom. Hodnota referencie, výstupu a akčného člena sa zaznamená a celý postup sa opakuje až do času, kedy doba trvania programu nepresiahne nastavenú hodnotu. Ako náhle program trvá dlhšie ako je požadované, pumpa sa deaktivuje, výsledné zaznamenané hodnoty sa uložia do súboru `response.mat` a graficky sa vykreslia.

```

y = (PressureShield.sensorRead()-init)/100; % [hPa] OverPressure

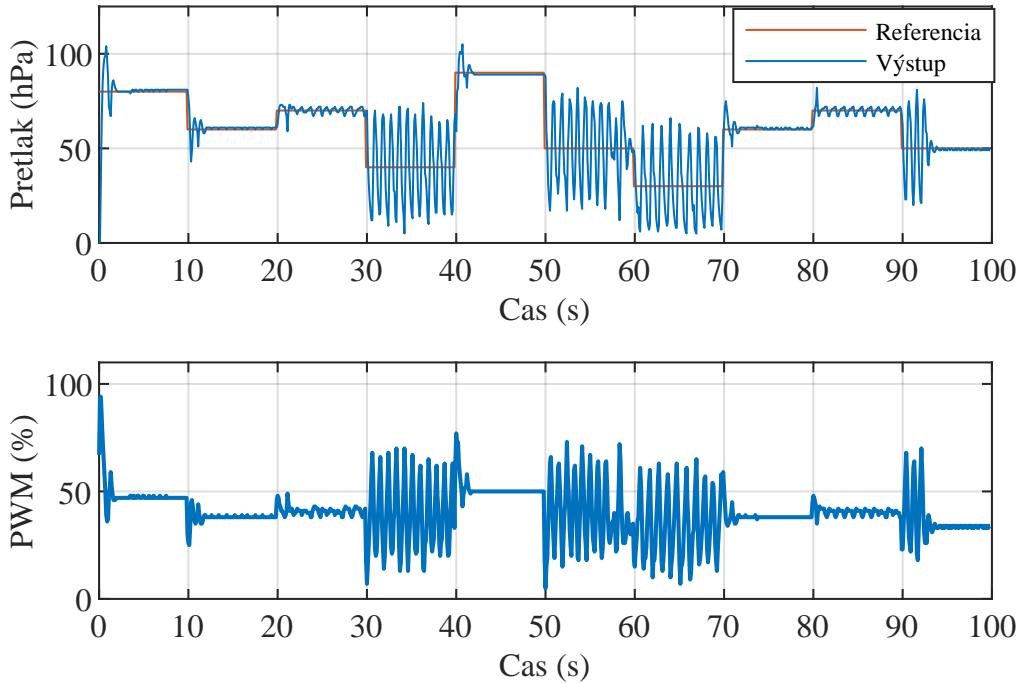
u = PID.compute(r-y, 0, 100, 0, 100);

PressureShield.actuatorWrite(u);    % [%] Power
response(k,:) = [r y u];          % Store results

```

Na Obr. 4.6 je zobrazená referencia, vstup a výstup experimentu PID regulátora pusteného v MATLABe. Na prvý pohľad si možno všimnúť, že sa počet vzoriek oproti príkladu v C++ zmenil. Tento jav je spôsobený tým, že funkcia `sensorRead` v MATLABe je pomerne pomalá kvôli načítavaniu registrov a kvôli operáciám s načítanými bitmi, čoho následkom je trvanie funkcie 100 ms, čo je desať násobok pôvodného vzorkovania 10 ms. Celá vzorka trvá 125 ms. Z tohto dôvodu som sa rozhodol nastaviť vzorkovanie tohto príkladu PID na 200 ms. So zmenou času vzorkovania som zároveň musel zmeniť aj regulačné konštanty. Hodnoty použité v tomto príklade sú $K_p = 0.45$, $T_i = 0.25$ a $T_d = 0.01$. Rovnako ako v predošлом príklade, aj v tomto prípade sa prejavuje nelinearita systému, čo znamená, že čím je nižšia hodnota pretlaku v nádobe, tým menej plynulá je regulácia. V prípade MATLABu je však tento jav oproti príkladu v C++ viditeľnejsí kvôli

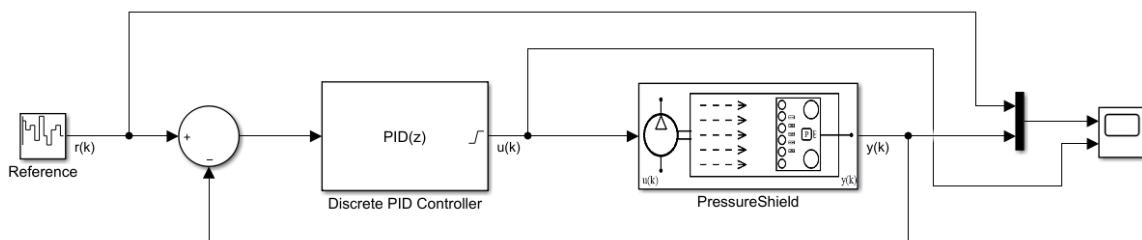
pomalšiemu vzorkovaniu. Hranica, kedy začne byť regulácia plynulejšia je okolo hodnoty 50 hPa.



Obr. 4.6: Priebeh vstupu a výstupu PID regulátora v MATLABe.

4.3.3 PID riadenie pre Simulink

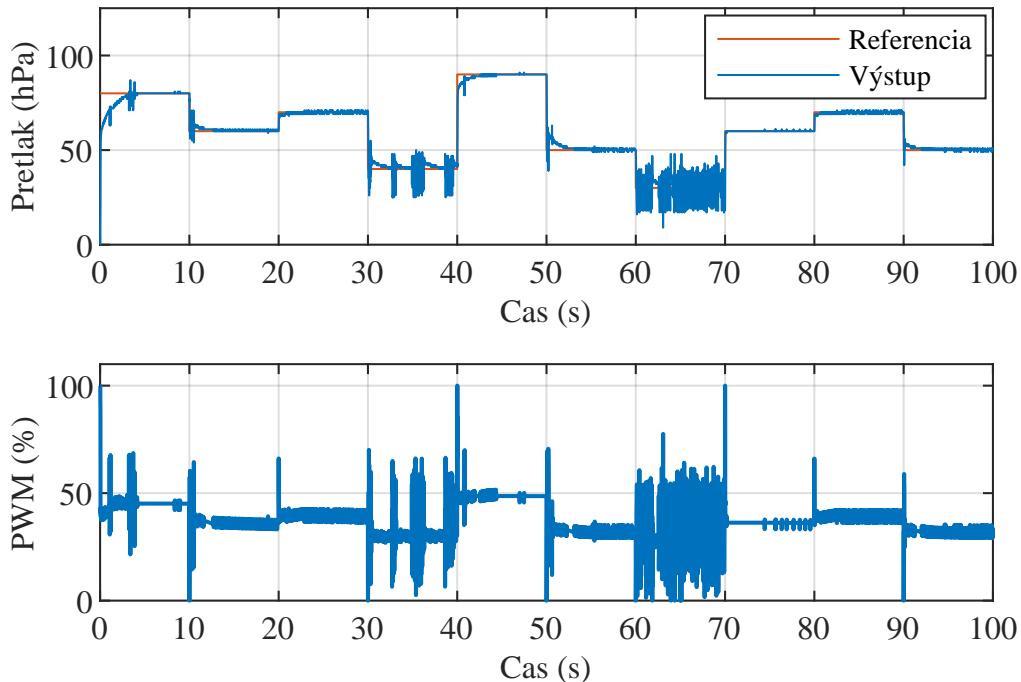
Posledným PID regulátorom, ktorý som vytvoril, je regulátor v Simulinku s názvom `PressureShield_PID.slx`. Celkový princíp je rovnaký ako v predošlých dvoch prípadoch. V tomto prípade je však možné niektoré výpočty, ktoré bolo v predošlých prípadoch potrebné naprogramovať, nahradieť blokmi Simulinku. Schéma príkladu je zobrazená na Obr. 4.7.



Obr. 4.7: Simulink PID regulátor.

Prvým blokom v tejto schéme je blok `Reference`. V tomto bloku sa nastavuje referencia ako vektor s požadovanými hodnotami. Taktiež sa tu nastavuje dĺžka úseku jednej referencie, v tomto prípade 10 sekúnd. Ďalší blok vykonáva výpočet odchýlky $e(t)$

medzi žiadoucou referenciou a výstupom systému. Blok Discrete PID Controller je blok Simulinku, v ktorom sa nastavujú regulačné konštanty PID regulátora. Blok PressureShield je vytvorené API, ktoré spracuje vypočítanú hodnotu akčného člena vychádzajúcu z PID bloku, odmeria hodnotu tlaku v nádobe a na výstupe zapíše hodnotu prettlaku v nej. Hodnota referencie, akčného člena a výstupu sa nakoniec vykreslia pomocou bloku Scope.



Obr. 4.8: Priebeh vstupu a výstupu PID regulátora v Simulinku.

Vykreslená referencia, vstup a výstup systému je zobrazená na Obr. 4.8. V prípade regulácie v Simulinku bolo potrebné taktiež zadať iné hodnoty ako v predošlých príkladoch, pretože aj v tomto prípade je doba vzorkovania odlišná. Hodnoty, ktoré som si zvolil sú $K_p = 1.5$, $T_i = 1$ a $T_d = 0.01$. Vplyv nelinearity na systém zostal zachovaný, plynulosť regulácie sa výrazne zlepšuje pri prekročení hodnoty 50 hPa.

4.4 LQ riadenie

Po naprogramovaní príkladov PID regulátorov som začal tvoriť druhý typ spätnoväzbovej regulácie. V tomto prípade sa jedná o lineárno-kvadratický (skrátene LQ) regulátor. Jedná sa o komplexnejší druh regulácie ako v prípade PID, keďže na fungovanie LQ je potrebné mať systém matematicky opísaný pomocou matíc modelu, ktorý som vytvoril v rámci identifikácie a modelovania.

Riadiaci spätnoväzbový zákon LQ regulátora je:

$$u_{(k)} = -K e_{(k)} = -K (X_{(k)} - X_{r(k)}) , \quad (4.3)$$

kde:

- $u_{(k)}$ je vypočítaný vstup systému v diskrétnom čase k ,
- \mathbf{K} matica LQ zosilnenia,
- $x_{(k)}$ je stavový vektor systému v diskrétnom čase k ,
- $x_{r(k)}$ je vektor referencie v diskrétnom čase k ,
- $e_{(k)}$ je odchýlka systému so zohľadnením referencie v diskrétnom čase k .

Na výpočet odchýlky systému je potrebné vedieť hodnotu jeho stavov. Pokial však hodnota stavov nie je odmeraná, využíva sa na ich odhad Kalmanov filter. Riadenie stavov prebieha pomocou matice zosilnenia \mathbf{K} , ktorá sa získava riešením Riccatiho rovnice. Matica zosilnenia sa počíta takým spôsobom, že riadiaci zákon LQ regulátora v prípade nulovej referencie minimalizuje kvadratickú účelovú funkciu

$$J_{(k)} = \sum_{k=1}^{\infty} (x_{(k)}^T Q x_{(k)} + u_{(k)}^T R u_{(k)}) , \quad (4.4)$$

kde:

- $J_{(k)}$ je kvadratická účelová funkcia,
- \mathbf{Q} je penalizačná matica stavov,
- \mathbf{R} je penalizačná matica vstupov.

Pomocou tejto rovnice sa dá numericky vyjadriť indikátor kvality riadenia, kde nižšia hodnota znamená kvalitnejšie riadenia a vysoká hodnota znamená nekvalitné riadenie. Na základe tejto definície, nájdenie minima funkcie znamená optimálne riadenie [12].

Môj spojity stavový model je potrebné zdiskretizovať, aby sa pomocou neho dali riadiť diskrétné procesy. Vyjadrenie diskrétneho stavového modelu je

$$x_{(k+1)} = Ax_{(k)} + Bu(k), \quad (4.5)$$

$$y_{(k)} = Cx_{(k)} + Du(k), \quad (4.6)$$

kde:

- $y_{(k)}$ je vektor výstupu systému v diskrétnom čase k ,
- \mathbf{A} je systémová matica,
- \mathbf{B} je matica vstupu,
- \mathbf{C} je matica výstupu,
- \mathbf{D} je priechodová matica (v našom prípade rovná 0) [12].

Pre model daný maticami \mathbf{A} a \mathbf{B} , ktorý má zvolené penalizačné matice \mathbf{Q} a \mathbf{R} by malo byť možné vypočítať hodnotu zosilnenia K . Na výpočet zosilnenia je však potrebná aj hodnota koncového stavu P . Na výpočet týchto hodnôt sa využíva Riccatiho algebrická rovnica

$$P_{(k)} = (A + BK_{(k)})^T P_{(k)} (A + BK_{(k)}) + K_{(k)}^T R K_{(k)} + Q, \quad (4.7)$$

$$K_{(k)} = (R + B^T P_{(k)} B)^{-1} B^T P_{(k)} A, \quad (4.8)$$

Na uľahčenie výpočtu týchto rovníc sa dá využiť MATLAB príkaz `dlqr`.

Kvôli možnosti sledovať referenciu je potrebné do LQ regulácie zaviesť tzv. integračnú zložku. Aktuálna odchýlka riadenia sa vypočíta ako

$$x_{(k+1)}^I = x_{(k)}^I + (r_k - y_k), \quad (4.9)$$

čo po dosadení za výstupy dá rovnicu

$$x_{(k+1)}^I = x_{(k)}^I + (r_k - Cx_k), \quad (4.10)$$

Na získanie rozšírenej stavovej reprezentácie je potrebné skombinovať integrátor z Rov. 4.10 s dynamickým modelom systému. Rozšírený model predikcie obsahujúci integrátor má tvar

$$\begin{bmatrix} x_{(k+1)} \\ x_{(k+1)}^I \end{bmatrix} = \begin{bmatrix} A & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} x_{(k)} \\ x_{(k)}^I \end{bmatrix} + \begin{bmatrix} B \\ 0 \end{bmatrix} u_{(k)} + \begin{bmatrix} 0 \\ I \end{bmatrix} r_{(k)}. \quad (4.11)$$

Pokiaľ sa označí rozšírený stav ako \tilde{x}_k , model rozšírený o integrátor bude mať tvar

$$\tilde{x}_{(k+1)} = \tilde{A}\tilde{x}_{(k)} + \tilde{B}u_{(k)} + r_{(k)}, \quad (4.12)$$

kde:

- $\tilde{\mathbf{A}}$ je matica systému rozšírenej dynamiky,
- $\tilde{\mathbf{B}}$ je matica vstupu rozšírenej dynamiky [12].

4.4.1 LQ simulácia

Pred vytvorením príkladov LQ regulátorov som vytvoril MATLAB script so simuláciou nazvaný `PressureShield_LQ_Simulation`, ktorý som využil na výpočet LQ zosilnenia K . Ako prvé sa v simulácii načíta hodnota referencie zo súboru `Reference.mat`. Ďalej sa načíta BlackBox model zo súboru `BlackBoxModel.mat`. Načítaný model sa rozšíri o tretí integračný stav, ktorý slúži na sledovanie hodnoty referencie v rámci simulácie, taktiež sa využíva na výpočet zosilnenia.

```

AI = [A, zeros(2, 1); -C, 1];
BI = [B; 0];
CI = [C, 0];

```

Po pridaní integrátora sa zadefinujú hodnoty penalizačných matíc **R** a **Q** a pomocou MATLAB príkazu `dlqr`, do ktorého sa dosadia potrebné hodnoty, sa vypočítajú hodnoty LQ zosilnenia **K**.

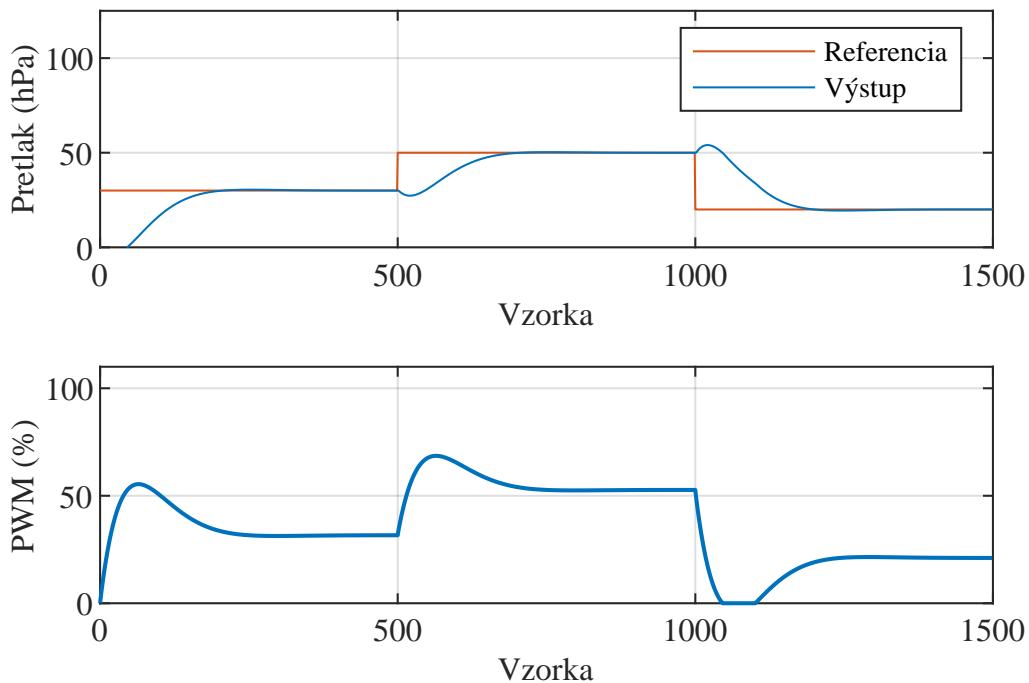
```

R=100;
Q=diag([1 10 0.01]);

K = dlqr(AI,BI,Q,R); %LQ gain calculation

```

V tomto prípade vyšli hodnoty $\mathbf{K} = (1.2501, 0.05, -0.01)$. V samotnej simulácii sa vyráta hodnota akčného člena $U_{(k)}$, pričom je na jeho hodnotu aplikovaná saturácia, ktorá obmedzuje jeho rozsah na hodnoty 0 – 100. Ďalej sa vypočíta hodnota stavového vektora $x_{(k+1)}$, hodnota vektora výstupu systému $y_{(k)}$ a pripočíta sa hodnota odchýlky. Nakoniec sa celý priebeh simulácie vykreslí. Vykreslený priebeh je zobrazený na Obr. 4.9. Celý kód simulácie je uvedený v dodatku B.5.



Obr. 4.9: Priebeh LQ simulácie.

4.4.2 LQ riadenie pre Arduino IDE

Po vytvorení simulácie, s ktorej pomocou sa vypočítali LQ zosilnenie **K**, začal som s tvorbou prvého príkladu LQ regulátora, konkrétnie regulátora v prostredí Arduino IDE. Na začiatku programu sa klasicky zahrňú potrebné knižnice, určí sa perióda vzorkovania, žiadana referencia pre prípad jej automatického určovania a dĺžka jednej sekcie automatickej

referencie. Na zadefinovanie matíc modelu, penalizačných matíc a matice zosilnenia som využil v prípade prostredia Arduino IDE knižnicu vytvorenú v rámci projektu AutomationShield s názvom

BasicLinearAlgebra.h

Táto (skrátene BLA) knižnica v sebe zahŕňa celú logiku zapisovania matíc v jazyku C++, užívateľ potom už len pomocou zadefinovaného zápisu určí rozmer, názov a hodnoty požadovanej matice.

```
BLA::Matrix<2, 2> A = {0.993, 0.01526, -0.003223, 0.4206};           // State
matrix A
BLA::Matrix<2, 1> B = {0.008751, -0.1501};                           // Input
matrix B
BLA::Matrix<1, 2> C = {1.74, 0.8755};                                     // Output
matrix C
```

Vyššie je uvedený príklad zápisu matíc modelu, ktorý som pre svoju LQ reguláciu využil. Rovnakým štýlom sa zapisujú aj penalizačné matice a matica LQ zosilnenia.

Inicializačná časť programu `setup` je v prípade LQ regulátora zhodná s touto časťou v PID regulátore. Zadefinuje sa v nej sériová komunikácia, zavolá sa inicializačná a kalibračná funkcia zariadenia PressureShield a určí sa períoda vzorkovania. Opakujúca sa časť programu `loop` sa taktiež zhoduje, kontroluje sa v nej či sa má vykonať ďalší krok programu, pokiaľ áno tak sa vykoná. Funkcia `step` vykonávajúca kroky programu najskôr kontroluje digitálny konektor 3 označený ako MS určujúci spôsob vyberania referenčnej hodnoty pretlaku. V prípade manuálneho nastavenia načítava hodnotu z potenciometra, v prípade automatického nastavenia načítava hodnotu z dopredu zadefinovaných referencií.

Po určení požadovanej referencie si program načíta hodnotu pretlaku v nádobe pomocou funkcie `sensorRead`. Načítanú hodnotu pretlaku, spolu so zadefinovanými maticami a hodnotou akčného člena, využije na výpočet odhadu stavového vektora pomocou Kalmanovho filtra. Tento výpočet sa uskutoční pomocou funkcie

PressureShield.getKalmanEstimate()

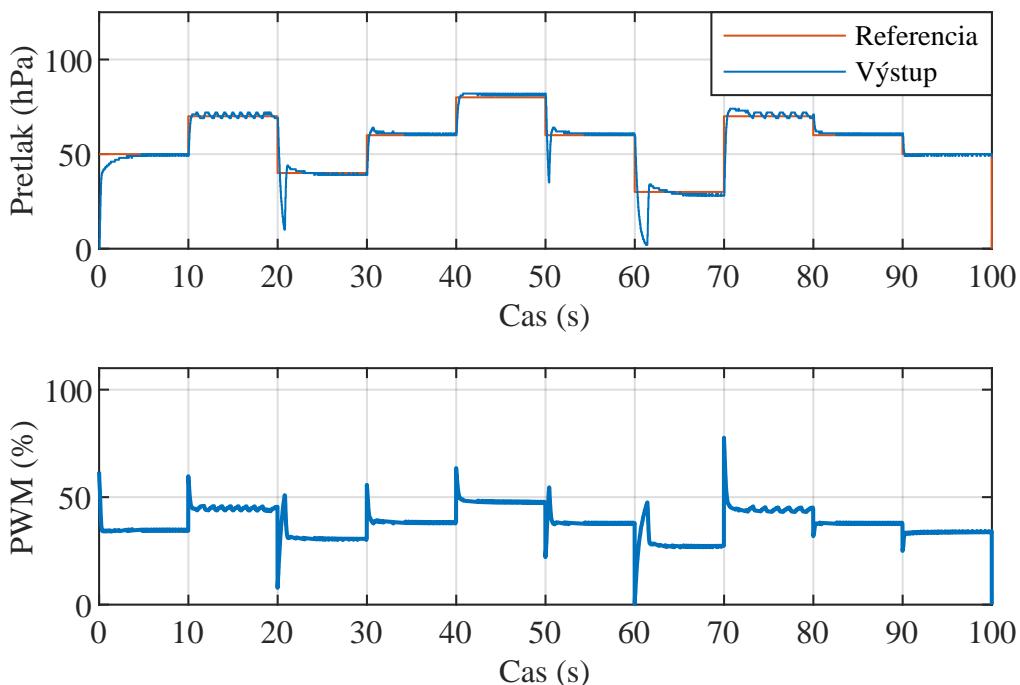
ktorá sa do programu načítava cez knižnicu `PressureShield.h`. Funkcia zahrnutá v súbore `getKalmanEstimate.inl` je C++ funkcia vytvorená pre AutomationShield slúžiaca na výpočet odhadu stavového vektora pomocou Kalmanovho filtra po dosadení potrebných hodnôt.

K hodnote stavového vektora sa ešte pripočíta hodnota odchýlky. V mojom prípade bolo potrebné od hodnoty odchýlky odrátať hodnotu referencie vynásobenú konštantou 0.36. Jedná sa o použitie tzv. Bulharskej konštanty. Tento krok bol potrebný kvôli nelinearite systému, ktorá spôsobovala, že tento LQ regulátor sa sice dal naladiť na reguláciu určitej hodnoty, s rastúcou referenciou od tejto hodnoty sa však zvyšovala aj regulačná odchýlka systému. Tento jav sa tak do určitej miery, vďaka úprave vzorca, dokázal vykompenzovať. Táto kompenzácia však nie je správnym riešením, keďže je lineárna. Z toho vyplýva, že bud' vstupuje do lineárnej časti LQ regulácie, kde v konečnom dôsledku kompenzuje výslednú nelinearitu, alebo len redukuje nelinearitu do takej miery, že nie je viditeľná. Posledným výpočtom je výpočet akčného člena u , ktorého hodnota sa nakoniec

zapíše do funkcie `actuatorWrite`.

```
y = PressureShield.sensorRead();
PressureShield.getKalmanEstimate(X, u, y, A, B, C, Q_Kalman, R_Kalman);
X(2) = X(2) + (Xr(0) - X(0) - Xr(0)*0.36);
u = -(K * (X - Xr))(0);
PressureShield.actuatorWrite(u);
```

Hodnota referencie, výstupu a vstupu systému sa následne vykreslia pomocou sériového portu. Celý kód LQ regulátora pre prostredie Arduino IDE je uvedený v dodatku A.4.

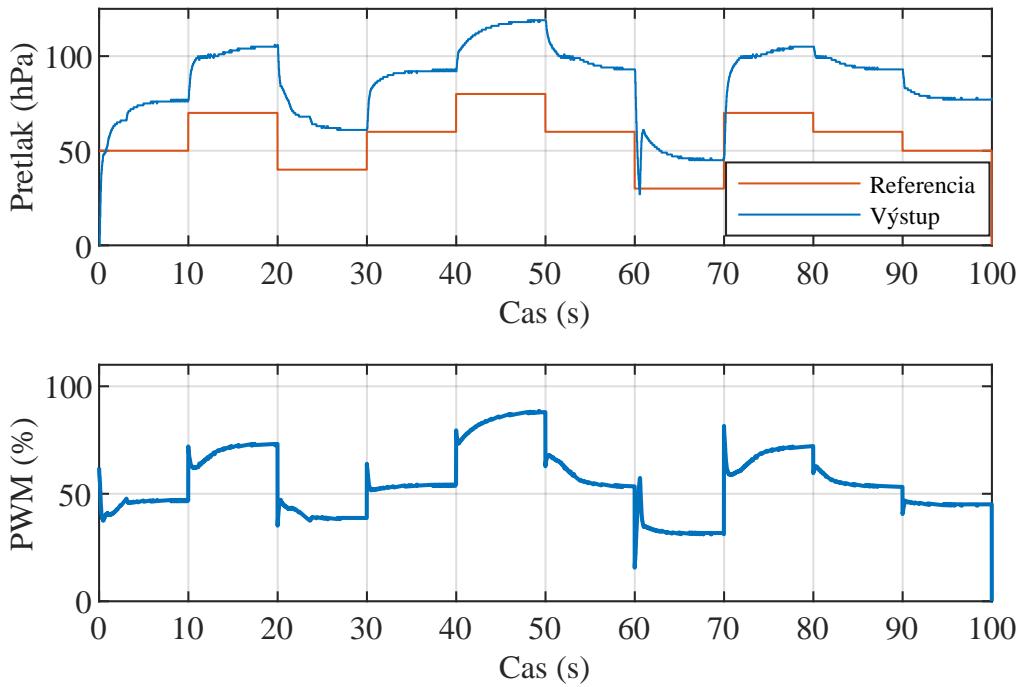


Obr. 4.10: Priebeh vstupu a výstupu LQ regulátora v Arduino IDE.

Na Obr. 4.10 možno vidieť, že aj napriek nelinearite systému, využitá kompenzácia regulačnej odchýlky pomocou odčítania referencie prenásobenej konštantou 0.36 bola pomerne efektívna. Hodnotu tejto konštanty som získal pomocou pokusov typu pokus a omyl, konkrétnie pomocou pozorovania výstupu LQ regulácie pri odčítavaní hodnoty referencie, vynásobenej rôznymi konštantami, od hodnoty odchýlky systému. Pre porovnanie je uvedený priebeh LQ regulácie bez tejto kompenzácie na Obr. 4.11. Na tomto obrázku je možné všimnúť si, že odchýlka sa skutočne mení so zmenou referencie.

4.4.3 LQ riadenie pre MATLAB

Príklad LQ regulátora sa v tomto vývojovom prostredí výrazne nelísi od regulátora v prostredí Arduino IDE. V prvom príklade LQ regulátora som dokázal odhaliť určité vlastnosti systému, ktoré spôsobujú čiastočné problémy, pokiaľ je systém ovládaný týmto typom regulácie. Tieto poznatky sa ďalej využili pri tvorbe programu v prostredí MATLAB, ktorý



Obr. 4.11: Priebeh vstupu a výstupu LQ regulátora bez kompenzácie.

má názov

`PressureShield_LQ.m`

Rovnako ako všetky predošlé programy, ktoré sa ďalej nahrávali na GitHub, aj tento začína kvôli CI inicializačnou funkciou `startScript`. Ďalej sa rovnako ako v prípade PID regulátora vytvorí objekt `PressureShield` vychádzajúci z triedy vytvorennej v našom API, zadefinuje sa periódā vzorkovania, hodnoty referencie v prípade automatického chodu programu, dĺžka jednej sekcie referencie a minimálna a maximálna možná hodnota akčného člena u , ktoré slúžia na saturáciu vstupu systému. V rámci inicializácie programu sa taktiež zadefinujú matice BlackBox modelu, penalizačné matice a matica LQ zosilnenia.

```
% System state-space matrices
matA = [0.993, 0.01526; -0.003223, 0.4206];
matB = [0.008751; -0.1501];
matC = [1.74, 0.8755];

% Penalisation matrices
Q_kalman = [0.05, 0; 0, 0.59];           % State penalisation
R_kalman = 1;                             % Input penalisation

% Calculate LQ gain that includes integrator state
K = [1.2501, 0.05, -0.01];
```

Ako už bolo spomenuté v PID príklade, na rozdiel od kódu v C++, MATLAB API nemá vyriešenú úpravu výstupu funkcie `sensorRead` na hodnotu pretlaku v nádobe.

Kvôli tomuto sa do premennej `init` pred spustením cyklu uloží hodnota atmosférického tlaku v nádobe pred jej natlakovaním.

Samotný cyklus prebieha pomocou funkcie `while`. Na začiatku cyklu sa určuje hodnota referencie a kontroluje sa, či sa jej hodnota nemá zmeniť na ďalšiu v poradí. V tomto úseku sa taktiež kontroluje, či už neboli zadané všetky hodnoty referencie. Pokiaľ áno, vypne sa pumpa a celý cyklus sa ukončí. Po vyriešení krokov týkajúcich sa referencie program načíta hodnotu tlaku v nádobe, odpočíta od nej premennú `init` a výsledok vydelí číslom 100. Vďaka tomuto sa zapíše do premennej `y` hodnota pretlaku v nádobe v hPa. Na výpočet odhadu stavového vektora pomocou Kalmanovho filtra sa využíva aj v MATLABe funkcia vytvorená v rámci projektu AutomationShield s názvom `estimateKalmanState`. Ďalej sa musí aj v tomto prípade pri pripočítaní odchýlky systému odrátať hodnota referencie vynásobená konštantou 0.36, pretože aj na tento LQ regulátor pôsobili rovnaké dôsledky nelinearity ako na príklad v C++. Výsledné hodnoty sa využijú na výpočet hodnoty akčného člena `u`, ktorá sa nakoniec upraví na rozsah 0 –100 a zapíše do funkcie `actuatorWrite`.

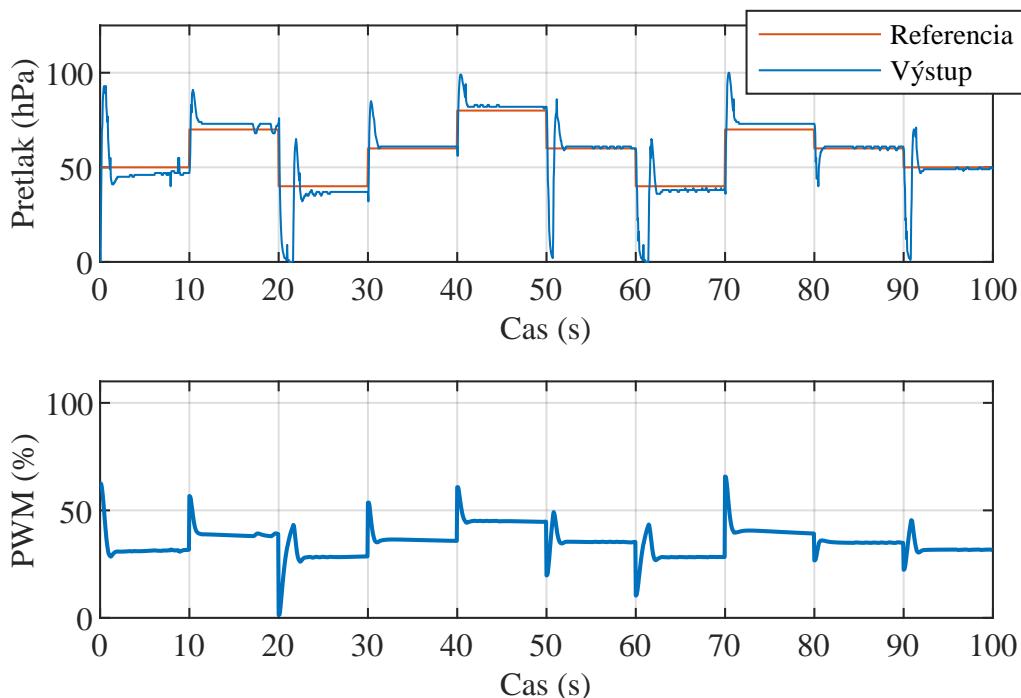
```

y = double((PressureShield.sensorRead()-init)/100); % Read overpressure

X(1:2) = estimateKalmanState(u, y, matA, matB, matC, Q_kalman, R_kalman);
X(3) = X(3) + (Xr(1) - X(1) - Xr(1)*0.36);

u = -K * (X - Xr); % Calculate LQ system input
u = constrain(u,umin,umax);
PressureShield.actuatorWrite(u); % Actuate

```



Obr. 4.12: Priebeh vstupu a výstupu LQ regulátora v MATLABe.

Hodnota referencie, vstupu a výstupu sa zapisujú každý cyklus do poľa s názvom

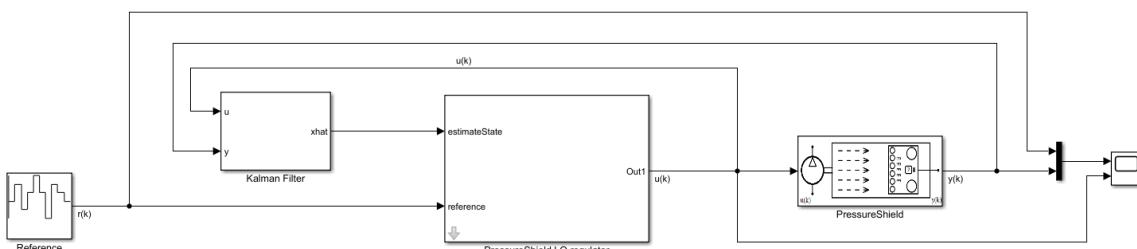
response, pomocou ktorého sa na konci programu vykreslí ich priebeh v čase. Celý kód MATLAB LQ regulátora je zobrazený v dodatku B.6.

Na Obr. 4.12 je tento priebeh znázornený. Stále platí, že rýchlosť tohto programu je obmedzená kvôli relatívne dlhej dobe trvania MATLAB funkcie `sensorRead`, z tohto dôvodu je na priebehu vykreslených menej vzoriek ako v prípade priebehu v Arduino IDE za rovnaký čas. Kvôli tejto vlastnosti MATLAB API je taktiež pomalšia reakcia systému na zmenu referencie a trvá mu dlhšie, kým sa regulovaná hodnota tlaku ustaľí.

Vplyv hodnoty referencie vynásobenej konštantou 0.36 a následne odčítanej od odchýlky systému je rovnaký ako v prvom príklade LQ regulátora, bez tejto úpravy vzorca by sa vytvárala regulačná odchýlka, ktorá by sa zvyšovala so zvyšujúcou sa referenciou. Keďže rozdiel medzi priebehom s úpravou a bez nej je veľmi podobný ako v prípade regulátora v C++, priebeh bez úpravy už v tomto prípade nie je zobrazený.

4.4.4 LQ riadenie pre Simulink

Simulink schéma `PressureShield_LQ.slx` je posledným príkladom LQ regulácie, ktorý som v rámci tejto práce vytvoril. Schéma LQ regulátora vytvoreného v Simulinku je zobrazená na Obr. 4.13.



Obr. 4.13: Simulink LQ regulátor.

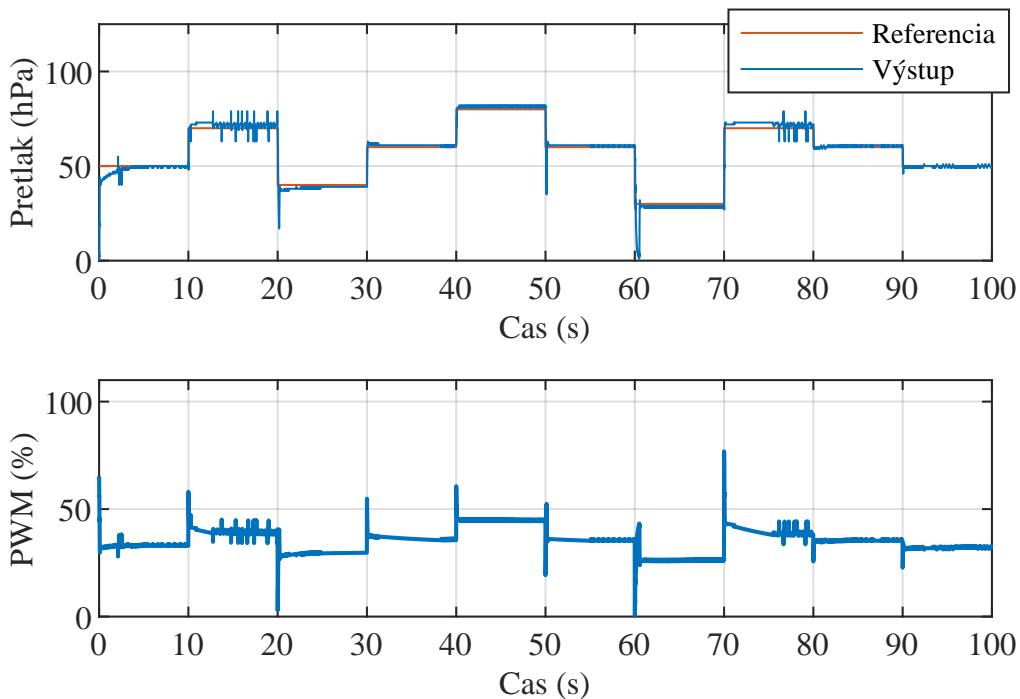
Referencia sa aj v tomto prípade Simulink schémy určuje pomocou bloku `Reference`. Mnou nastavené hodnoty sú $\mathbf{R} = [50 \ 70 \ 40 \ 60 \ 80 \ 60 \ 30 \ 70 \ 60 \ 50]$, čas jednej sekcie referencie sa nastavuje na 10 sekúnd. Tento blok sa následne napája na blok s názvom `PressureShield LQ regulator`. Okrem referencie vstupuje do tohto bloku aj vypočítaný odhad stavového vektora pomocou Kalmanovho filtra, a do masky bloku sa nastavuje vektor LQ zosilnenia a períoda vzorkovania. V samotnom bloku sa nachádza MATLAB funkcia, ktorá slúži na výpočet odchýlky regulácie a na výpočet vstupu systému u .

```
function u = fcn(x, r, K)
persistent xI;
if isempty(xI)
    xI = 0;
end

xI = xI + r - x(1) - r*0.36;
x = [x; xI];
u = -K * (x - [r; 0; 0]);
```

Ako je možno vidieť v kóde uvedenom vyššie, aj v prípade LQ regulátora v prostredí Simulink, rovnako ako v dvoch predošlých prípadoch, bolo potrebné kvôli nelinearite systému odpočítať od výslednej odchýlky regulácie referenciu vynásobenú konštantou 0.36. Výstupom z bloku je hodnota vstupu systému u , ktorá ďalej vstupuje do bloku PressureShield, ktorý aktivuje pumpu s potrebným PWM signálom a zároveň načíta hodnotu pretlaku v nádobe.

Na výpočet odhadu stavového vektora sa pomocou Kalmanovho filtra používa v tomto príklade Simulink blok s názvom Kalman Filter. Vstupom do bloku sú vstup systému $u_{(k)}$ a výstup systému $y_{(k)}$. V samotnom bloku sa zadefinovali matice modelu a penalizačné matice. Rovnako ako pri každom z príkladov, ktoré som vytvoril, aj v tomto prípade sa penalizačné matice líšia. Konkrétnie v tomto LQ regulátore je ich hodnota $\mathbf{Q} = \begin{pmatrix} 5 & 0 \\ 0 & 1 \end{pmatrix}$ a $\mathbf{R} = (1)$. Výstup z bloku, t.j. odhad stavového vektora, vstupuje do bloku PressureShield LQ regulator. Celý priebeh referencie, vstupu a výstupu systému sa dá zobraziť pomocou bloku Scope umiestneného na konci schémy. Tento priebeh je zobrazený na Obr. 4.14.



Obr. 4.14: Priebeh vstupu a výstupu LQ regulátora v Simulinku.

Na priebehu je možné si všimnúť, že kompenzácia nelinearity bola aj v tomto prípade úspešná. Regulátor je veľmi stabilný, a drží približne hodnotu referencie aj pri nízkej referencii 30hPa, aj pri vysokej referencii 80hPa. Taktiež prekmit vstupu systému nie je taký výrazný ako v prípade LQ regulátora v MATLABe, a systém sa dokáže veľmi efektívne rýchlo ustabilizovať.

4.5 MPC riadenie

MPC regulátor je najpokročilejší druh regulátora z troch typov regulátorov, ktorými som sa v tejto práci zaoberal. Pointou MPC regulácie je regulovanie systému v súčasnosti za pomocí predikcie budúcich stavov. Na rozdiel od PID a LQ regulátora má hranice vstupu obmedzené. Nevýhoda tohto regulátora je v jeho zložitých matematických operáciách. Z tohto dôvodu môže byť jeho použitie veľmi náročné na výpočtový hardvér.

Rovnica riešenia MPC problematiky je

$$\vec{u}_{(k)}^* = \underset{\vec{u}_{(k)}}{\operatorname{argmin}} \left(\frac{1}{2} \vec{u}_{(k)}^T \mathbf{H} \vec{u}_{(k)} + \mathbf{x}_{(k)}^T \mathbf{G}^T \vec{u}_{(k)} \right), \quad (4.13)$$

kde:

- $\vec{u}_{(k)}^*$ je optimálna postupnosť systémových vstupov v diskrétnom čase k ,
- $\vec{u}_{(k)}$ je postupnosť systémových vstupov v diskrétnom čase k ,
- $\mathbf{x}_{(k)}$ je stavový vektor v diskrétnom čase k ,
- \mathbf{H} je Hessova matica (Hessián),
- \mathbf{G} je matica MPC účelovej funkcie, pričom platí $\mathbf{g}^T = \mathbf{x}_k^T \mathbf{G}^T$.

Podľa rovnice 4.13 sa MPC riadenia snaží nájsť ideálnu postupnosť vstupov, ktoré by zminimalizovali účelovú funkciu. Dĺžka postupnosti závisí od predikčného horizontu n_p , ktorý určuje, koľko dopredu sa má funkcia minimalizovať, resp. koľko budúcich vzoriek sa má brať do úvahy [12]. V rovnici 4.13 je predikčný horizont zahrnutý v maticiach \mathbf{H} a \mathbf{G} . Hodnota účelovej funkcie MPC je definovaná ako

$$J_{(k)} = \vec{u}_{(k)}^T \mathbf{H} \vec{u}_{(k)} + 2\mathbf{x}_{(k)}^T \mathbf{G}^T \vec{u}_{(k)} + \mathbf{x}_{(k)}^T \mathbf{F} \mathbf{x}_{(k)}, \quad (4.14)$$

kde \mathbf{F} je matica účelovej funkcie, ktorá nemá vplyv na optimalizovanú premennú. Rovnica 4.13 je zjednodušeným tvarom rovnice 4.14, a keďže rovnica 4.13 hľadá optimálne vstupy, môžeme v nej maticu \mathbf{F} zanedbať. Matice \mathbf{H} , \mathbf{G} a \mathbf{F} sa dajú získať pomocou rekurzívnej iteračnej metódy za použitia matíc \mathbf{A} , \mathbf{B} , \mathbf{Q} , \mathbf{R} a predikčného horizontu n_p . Keďže sa matice \mathbf{H} , \mathbf{G} a \mathbf{F} v čase nemenia, stačí ich vypočítať raz na začiatku predikcie [12].

Ďalšími dôležitými parametrami sú matice \mathbf{A}_c , \mathbf{b}_0 a \mathbf{B}_0 . Tieto matice poskytujú vstupné a stavové obmedzenia, pomocou nerovnice

$$\mathbf{A}_c \vec{u}_{(k)} \leq \mathbf{b}_c, \quad (4.15)$$

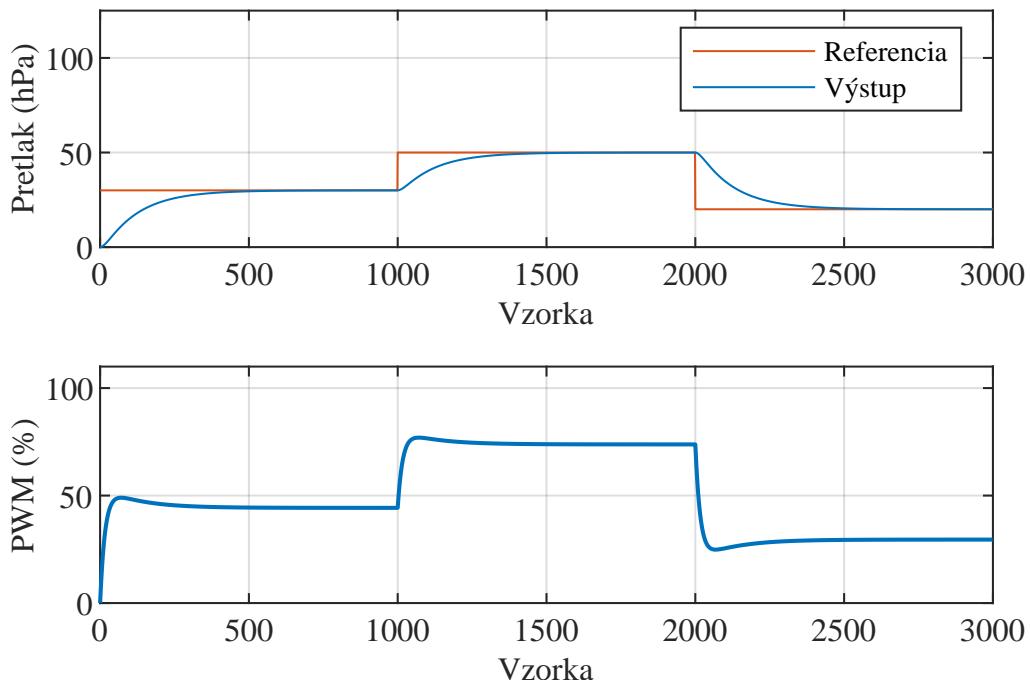
kde

$$\mathbf{b}_c = \mathbf{b}_0 + \mathbf{B}_0 \mathbf{x}_{(k)}. \quad (4.16)$$

Nerovnica 4.15 predstavuje matematickú podmienku riešenia procesu minimalizácie. Pomocou nej je možné nastaviť obmedzenie vstupu systému s použitím matíc \mathbf{a}_c a \mathbf{b}_0 alebo obmedzenie stavov systému pomocou matíc \mathbf{a}_c , \mathbf{b}_0 a \mathbf{B}_0 . Obidve obmedzenia môžu byť nastavené súčasne. Výpočet týchto troch matíc taktiež prebieha pomocou rekurzívnej iteračnej metódy. V prípade, že sa obmedzuje iba vstup systému, stačí tieto matice vypočítať raz na začiatku predikcie. Pokiaľ sa obmedzujú aj stavy systému, je potrebné aby došlo k násobeniu $\mathbf{B}_0 \mathbf{x}_{(k)}$ kvôli zmenám stavového vektora [12].

4.5.1 MPC simulácia

Rovnako ako v prípade LQ regulátora, aj pri MPC regulácii som vytvoril simuláciu celého riadenia v prostredí MATLAB s názvom PressureShield_MPC_Simulation.m. Simuluje sa v nej výstup systému vzhľadom na nastavenú referenciu a hodnota vstupu systému. Na začiatku sa načíta súbor s referenciou Reference.mat a BlackBox model zo súboru BlackBoxModel1.mat. Model sa rozšíri o tretí integračný stav a určia sa hodnoty penalizačných matíc kvôli výpočtu matice zosilnenia K a matice koncového stavu P. Ďalej sa zadefinuje ohraničenie vstupu a predikčný horizont n_p . Na výpočet matíc \mathbf{H} a \mathbf{G} sa využíva funkcia ucelova funkcia.m, ktorá bola vytvorená pri príklade simulácie MPC riadenia v [12]. Na výpočet obmedzení som taktiež použil funkciou z [12], konkrétnie funkciu obmedzenia.m. Obidve tieto funkcie sú k dispozícii na GitHube [15]. Script pomocou MATLAB funkcie quadprog, do ktorej sa dosádzajú matice \mathbf{H} a \mathbf{G} , a obmedzenia A_c a b_c , vypočíta postupnosť vstupov systému a priráta hodnotu do stavového vektora $\mathbf{x}_{(k)}$. Výsledky sa nakoniec vykreslia.



Obr. 4.15: Priebeh MPC simulácie.

Vykreslený priebeh simulácie je zobrazený na Obr. 4.15. V rámci priebehu je možné

všimnúť si, že systému opísanému BlackBox modelom, ktorý bol vytvorený pre zariadenie PressureShield, trvá v simulácii pomerne dlho, kým dosiahne požadovanú hodnotu pretlaku.

4.5.2 MPC riadenie pre MATLAB

Jediný príklad MPC regulátora, ktorý som vytvoril, bol vytvorený v prostredí MATLAB. Script `PressureShield_MPC.m` sa začína kvôli CI na GitHube funkciou `StartScript`. Ďalej sa rovnako ako v predošlých príkladoch vytvorí objekt `PressureShield`. Rovnako sa zadefinujú všetky konštanty potrebné pre chod programu. Minimálna a maximálna možná hodnota akčného člena u v tomto prípade slúžia na obmedzenie vstupu. V rámci inicializácie sa taktiež zadefinujú všetky potrebné matice a predikčný horizont n_p . Na výpočet matice zosilnenia \mathbf{K} a matice koncového stavu \mathbf{P} sa využije príkaz `dlqr`. Získanie hodnôt matíc \mathbf{H} a \mathbf{G} prebieha pomocou funkcie `getCostFunctionMPC`, ktorá bola vytvorená v rámci iniciatívy `AutomationShield`. Funkcia využitá na výpočet obmedzení `setConstraintsMPC` je taktiež jej súčasťou.

```
[K, P] = dlqr(matAhat, matBhat, Q, R);
% Get Hessian and Gradient
[H,G]=getCostFunctionMPC(matAhat,matBhat,np,Q,R,P);
% Set constraints
[Ac b0]=setConstraintsMPC(uL,uU,np);
```

Pred začatím cyklu sa ešte uloží hodnota tlaku v nádobe pred reguláciou do premennej `init`.

Samotný cyklus je podobný ako v prípade LQ regulátora. Po určení referencie sa načíta hodnota tlaku v nádobe a odčíta sa od nej hodnota premennej `init`. Funkciou `quadprog` sa vypočíta optimálna postupnosť vstupov systému a prvá hodnota z tejto postupnosti sa zapíše do funkcie `actuatorWrite`, ktorá podľa nej upraví výkon pumpy. Na odhad stavu sa v tomto prípade taktiež používa Kalmanov filter prostredníctvom funkcie `estimateKalmanFilter`. Keďže sa typ výpočtu odhadu ani nelinearita systému zmenou regulátora nezmenili, aj v MPC riadení musíme odrátať od hodnoty odhadu referenciu vynásobenú konštantou 0.36.

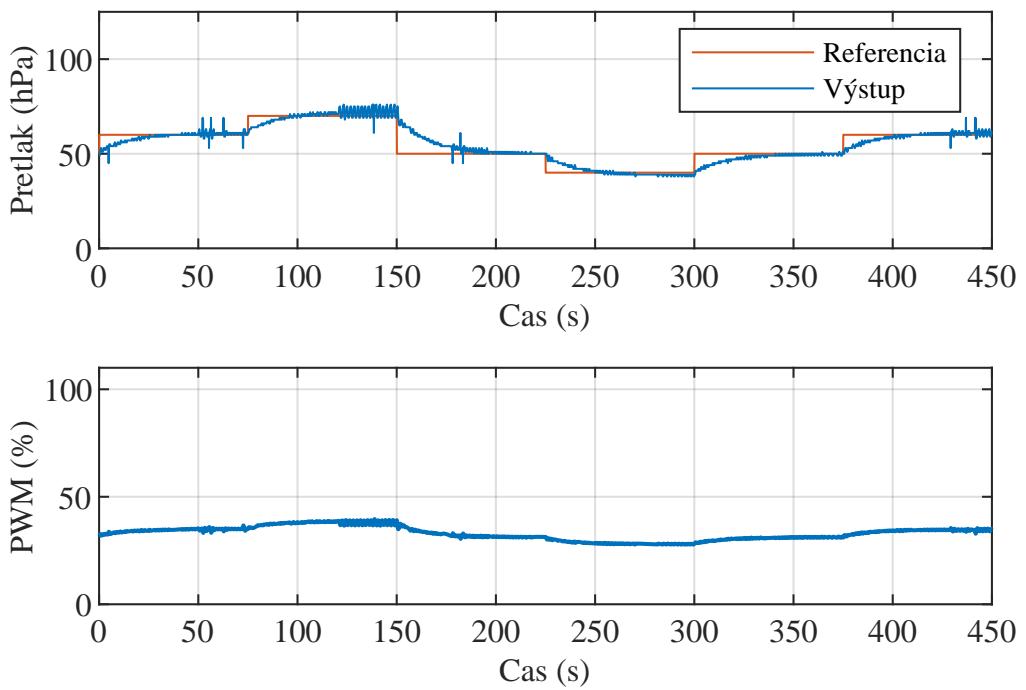
```
u(:, 1) = quadprog(H, G*X(:, 1), Ac, b0, [], [], [], [], opt);
PressureShield.actuatorWrite(u(1, 1));
X(1:2) = estimateKalmanState(u(1, 1), y, matA, matB, matC, Q_Kalman,
R_Kalman);
X(3) = X(3) + (Xr(1) - X(1) - Xr(1)*0.36);
```

Hodnota referencie, vstupu a výstupu sa uložia do premennej `response` a celý cyklus sa opakuje. Po ukončení programu sa priebeh regulácie vykreslí.

Celý priebeh MPC regulácie je zobrazený na Obr. 4.16.

Na tomto priebehu je viditeľné, že presne ako v simulácii, hodnote výstupu trvá dlhú dobu, kým sa dostane na referenčnú hodnotu. Taktiež na začiatku regulácie trvá dlhšiu dobu ktorú sa systém ustabilizuje. Keď však dosiahne referenciu, drží jej hodnotu bez výraznejšieho prekmitu alebo odchýlky.

Počas vývoja a testovania každého z mojich kódov som využíval platformu GitHub. Ako už bolo v texte spomenuté, na tejto platforme sa nachádzajú všetky kódy a knižnice, ktoré



Obr. 4.16: Priebeh vstupu a výstupu MPC regulátora v MATLABe.

boli vytvorené v rámci iniciatívy AutomationShield [14]. Pri tejto práci bola v rámci AutomationShieldu vytvorená vetva (angl. branch) s názvom PressureShield, aby mnou nahrané kódy nezasahovali do hlavnej vetvy. Počas tohto vývoja bol každý kód po pridaní skontrolovaný pomocou CI. Princíp tejto kontroly spočíva v overení, či je možné nahraný kód skompilovať. Taktiež kontroluje kompatibilitu s vybranými vývojovými doskami a obsahuje tiež kontrolu pravopisu komentárov. Toto CI je však stále vo vývoji, takže jeho kontrola nie je spoľahlivá na 100 %.

5 Záver

V rámci mojej práce som najskôr vytvoril schému zapojenia komponentov zariadenia PressureShield spolu s návrhom dosky plošných spojov. Následne som už vyrobenú dosku skompletoval. Vďaka veľkému počtu testov, počas ktorých doska nemala žiadne hardvérové problémy, môžem prehlásiť, že môj návrh a kompletovanie boli úspešné a doska je z hardvérového hľadiska plne funkčná. Taktiež vďaka tomu môžem poznamenať, že mnou vybrané súčiastky pre osadenie dosky sú vhodné a splňajú svoj účel, rovnako ako sú funkčné komponenty vytvorené pomocou 3D tlače.

Po vytvorení a otestovaní hardvéru som vytvorili API pre vývojové prostredia Arduino IDE, MATLAB a Simulink. V každom API bola vytvorená funkcia pre zápis hodnoty vstupu akčného člena `ActuatorWrite`, funkcia na čítanie referencie z potenciometra `ReferenceRead` a funkcia na načítanie hodnoty tlaku v nádobe pomocou senzora BMP280 s názvom `SensorRead`, v ktorej som na načítanie hodnôt využili načítanie údajov z registrov senzora pomocou komunikačného protokolu I2C, ktoré sa následne upravili tak, aby výstupom bol aktuálny tlak v nádobe.

Identifikáciu systému som vykonal pomocou mnou vytvorených programov, ktorých výstupné údaje som pomocou identifikačného toolboxu v prostredí MATLAB premenil na BlackBox model opisujúci môj systém matematicky. Pri identifikačných pokusoch sa zistilo, že na systém pôsobí nelinearita spôsobená tým, že nevyužívam na únik tlaku žiadny akčný člen, iba prirodzený únik, ktorý sa s meniacim sa pretlakom v nádobe taktiež mení.

Na demonštráciu funkčnosti zariadenia pri spätnoväzbom riadení som vytvoril pre každé vývojové prostredie príklad PID regulátora a príklad LQ regulátora. Pre prostredie MATLAB bolo vytvorené aj MPC riadenie. Keďže každé prostredie načítava údaje z registrov senzora pomocou I2C protokolu rozdielne dlho, líšili sa aj periódy vzorkovania mnou vytvorených príkladov. Z tohto dôvodu, som musel každý z mojich regulátorov nalaďať pomocou iných parametrov. Veľký vplyv na ladenie mala aj už spomenutá nelinearita systému, ktorá sa musela hlavne prípade LQ regulátorov výrazne kompenzovať. Nepodarilo sa mi však splniť všetky ciele práce.

Prvým problémom môjho riešenia je tzv. bulharská konštanta, ktorou kompenzujem nelinearitu v prípade LQ regulácie. Kompenzácia týmto spôsobom je lineárna, takže v konečnom dôsledku by nemala účinne kompenzovať nelinearitu. Prvý dôvod, kvôli ktorému to môže fungovať je, že nelinearita sice stále ostáva zachovaná, vďaka tejto konštante sa však jej vplyv natoľko zmenší, že pri regulovaní dosiahneme žiadané hodnoty. Druhou možnosťou je, že táto lineárna kompenzácia vstupuje do nejakej lineárnej časti LQ regulácie, kde dokáže vykompenzovať výslednú nelinearitu regulátora.

Druhý problém môjho riešenia spočíva v nekompatibilite zariadenia PressureShield zostrojeného pre túto prácu s 3.3 V logikou napájania. Moje PCB je sice navrhnuté na

3.3 V logiku, pri snahe zostrojiť API pre Python na doske Adafruit Metro M4 s 3.3 V logikou som však zistil, že toto napätie nedokáže aktivovať pumpu. Dôvodom tejto chyby je, že táto hodnota napäťia nedokáže zopnúť MOSFET ovládajúci pumpu.

V rámci prípadnej budúcej práce s týmto zariadením ostáva veľa možností, čo opraviť a zlepšiť. Hlavnou úlohou v budúcnosti by mala byť úprava LQ regulátora tak, aby na presnú reguláciu nebola potrebná bulharská konštantá, ktorú som využil ja. V rámci opravy by sa taktiež mal zmeniť MOSFET IRF3710 za iný s nižším napäťím potrebným pre zopnutie, aby nenastávali problémy s aktiváciou pumpy pri 3.3 V logike dosky. Čo sa týka ďalšieho vývoja, bolo by vhodné v rámci mojich API vytvoriť taktiež príklady spätnoväzbovej regulácie pomocou MPC regulátora. Po zmene MOSFETu sa taktiež vytvára možnosť vytvorenia API v jazyku Python pomocou dosky Adafruit Metro M4.

V konečnom dôsledku však môžem zhrnúť, že doska PressureShield je plne funkčná a je plne využiteľná pre vzdelávacie, prípadne výskumné účely.

Literatúra

- [1] Alfian Ma'arif, Iswanto, Nia Maharani Raharja, Phisca Aditya Rosyady, Ahmad Raditya Cahya Baswara, Aninditya Anggari Nuryono. Control of dc motor using proportional integral derivative (pid): Arduino hardware implementation. In *2020 2nd International Conference on Industrial Electrical and Electronics (ICIEE)*, pages 74–78, Lombok, Indonesia, October 2020.
- [2] Ali R. Jalalvand, Mahmoud Roushani, Hector C. Goicoechea, Douglas N. Rutledge, Hui-Wen Gu. Matlab in electrochemistry: A review. In *Talanta*, pages 205–225, March 2019.
- [3] Alps Alpine. SSAJ120100 datasheet. Datasheet. Online. cit. 25.3.2021, <https://eu.mouser.com/datasheet/2/15/SSAJ-1370983.pdf>.
- [4] Bosch. BMP280 digital pressure sensor. Datasheet. Online., 2018. cit. 15.3.2021, <https://www.gme.sk/data/attachments/dsh.772-274.2.pdf>.
- [5] Dan Mihai. Designing an experimental platform for the air pressure in a small tank by digital control. In *2016 International Conference on Applied and Theoretical Electricity (ICATE)*, Craiova, Romania, October 2016.
- [6] Dareen K. Halim, Tang Chong Ming, Ng Mow Song, Dicky Hartono. Arduino-based IDE for Embedded Multi-processor System-on-Chip. In *2019 5th International Conference on New Media Studies (CONMEDIA)*, pages 135–136, Bali, Indonesia, October 2019.
- [7] David J. Esteves, Alexandra Moutinho, José Raul Azinheira. Stabilization and altitude control of an indoor low-cost quadrotor: Design and experimental results. In *2015 IEEE International Conference on Autonomous Robot Systems and Competitions*, pages 150–155, Vila Real, Portugal, April 2015.
- [8] Gergely Takács and Martin Gulán. AutomationShield. Wiki. Online., 2020. cit. 20.4.2021, <https://github.com/gergelytakacs/AutomationShield/wiki>.
- [9] Jingmin Du, Lu Tan, Cheng Wu. An introduction of dynamic air pressure controller with high precision. In *2015 International Conference on Fluid Power and Mechatronics (FPM)*, pages 440–443, Harbin, China, August 2015.

- [10] Martin Staroň, Anna Vargová, Martin Vríčan, Vladimír Kmeť, Lukáš Kavoň, Radoslav Gago, Eva Vargová, Tomáš Tužinský, Matúš Leginus. BlowShield. Wiki. Online., 2020. cit. 20.4.2021, <https://github.com/gergelytakacs/AutomationShield/wiki/BlowShield/>.
- [11] Mohsen Shiee, K. Arman Sharifi, Morteza Fathi, Farid Najafi. Air pressure control via sliding mode approach using an on/off solenoid valve. In *20th Iranian Conference on Electrical Engineering (ICEE2012)*, pages 857–861, Tehran, Iran, May 2012.
- [12] G. Takács and M. Gulán. *Základy Prediktívneho Riadenia*. Spektrum STU, Bratislava, 1. edition, 2018.
- [13] G. Takács, M. Gulán, J. Bavlna, R. Köplinger, M. Kováč, E. Mikuláš, S. Zarghoon, and R. Salní. HeatShield: a low-cost didactic device for control education simulating 3D printer heater blocks. In *Proceedings of the 2019 IEEE Global Engineering Education Conference (EDUCON)*, pages 374–383, Dubai, United Arab Emirates, April 2019.
- [14] G. Takács, P. Chmurčiak, R. Koplinger, T. Konkoly, G. Penzimger, M. Gulán, J. Kulhánek, L. Vadovič, M. Biro, E. Vargová, M. Vríčan, and J. Mihalík. AutomationShield. <https://github.com/gergelytakacs/AutomationShield>, 2021.
- [15] G. Takács and M. Gulán. Základy prediktívneho riadenia. <https://github.com/gergelytakacs/Zaklady-prediktivneho-riadenia/tree/master/kap07>, 2018.
- [16] Tibor Konkoly and Gergely Takács. MotoShield. Wiki. Online., 2019. cit. 20.4.2021, <https://github.com/gergelytakacs/AutomationShield/wiki/MotoShield/>.
- [17] Xuling Liu, Songjing Li. Control method experimental research of micro chamber air pressure via a novel electromagnetic microvalve. In *2017 4th International Conference on Information Science and Control Engineering (ICISCE)*, pages 921–925, Changsha, China, July 2017.

A Arduino IDE kód

A.1 PressureShield.h

```
#ifndef PRESSURESHIELD_H_                                // Include guard
#define PRESSURESHIELD_H_

#include <AutomationShield.h>                          // Include the main library
#include <Wire.h>                                       // Include the I2C protocol library
#include <Arduino.h>

// Sensor definitions

#define BMP280_DIG_T1_LSB_REG    0x88
#define BMP280_DIG_T1_MSB_REG    0x89
#define BMP280_DIG_T2_LSB_REG    0x8A
#define BMP280_DIG_T2_MSB_REG    0x8B
#define BMP280_DIG_T3_LSB_REG    0x8C
#define BMP280_DIG_T3_MSB_REG    0x8D
#define BMP280_DIG_P1_LSB_REG    0x8E
#define BMP280_DIG_P1_MSB_REG    0x8F
#define BMP280_DIG_P2_LSB_REG    0x90
#define BMP280_DIG_P2_MSB_REG    0x91
#define BMP280_DIG_P3_LSB_REG    0x92
#define BMP280_DIG_P3_MSB_REG    0x93
#define BMP280_DIG_P4_LSB_REG    0x94
#define BMP280_DIG_P4_MSB_REG    0x95
#define BMP280_DIG_P5_LSB_REG    0x96
#define BMP280_DIG_P5_MSB_REG    0x97
#define BMP280_DIG_P6_LSB_REG    0x98
#define BMP280_DIG_P6_MSB_REG    0x99
#define BMP280_DIG_P7_LSB_REG    0x9A
#define BMP280_DIG_P7_MSB_REG    0x9B
#define BMP280_DIG_P8_LSB_REG    0x9C
#define BMP280_DIG_P8_MSB_REG    0x9D
#define BMP280_DIG_P9_LSB_REG    0x9E
#define BMP280_DIG_P9_MSB_REG    0x9F

//PressureShield definitions
```

```

#ifndef SHIELDRELEASE           // Define release version of used
    hardware
#define SHIELDRELEASE 1        // Latest version by default
#endif

// Defining pins used by the PressureShield board
#if SHIELDRELEASE == 1
#define PRESSURE_UPIN 11        // Pump (Actuator)
#define PRESSURE_RPIN A0        // Potentiometer runner (Reference)
#define PRESSURE_MSPIN 3         // Manual Switch
#endif

#define BMP280_addr            0x76      // Sensor address

class PressureClass {           // Class for PressureShield device
public:
    // PressureShield public function
    void begin(void);          // Board
    initialisation - initialisation of pressure sensor, pin modes and
    variables
    void Step(void);           // Board
    void calibration(void);     // Board
    calibration - finding out the minimal and maximal values measured
    by pressure sensor
    void actuatorWrite(float);   // Write
    actuator - function takes input 0.0-100.0 and sets pump speed
    accordingly
    float referenceRead(void);    //
    Reference read - returns potentiometer position in percentual range
    0.0-100.0
    float sensorRead(void);      // Sensor
    read - returns the pressure in percentual range 0.0-100.0

    // Sensor public functions
    void readCoefficients();
    void begin_config();
    void measure_config();
    float readPressure();
    void readTemperature();

private:
    // PressureShield private
    float _minPressure;          // Variable for storing minimal
    distance measured by sensor in milimetres
    float _maxPressure;          // Variable for storing maximal
    distance measured by sensor in milimetres
    float _sensorPercent;         // Variable for percentual
    altitude of the ball in the tube 0.0-100.0
    bool _wasCalibrated;          // Variable for storing
    calibration status

```

```

// Sensor private
uint8_t readRegister(uint8_t RegSet);
uint32_t press_read();
uint32_t temp_read();

int64_t t_fine;
uint16_t dig_T1;
int16_t dig_T2;
int16_t dig_T3;

uint16_t dig_P1;
int16_t dig_P2;
int16_t dig_P3;
int16_t dig_P4;
int16_t dig_P5;
int16_t dig_P6;
int16_t dig_P7;
int16_t dig_P8;
int16_t dig_P9;
uint32_t temp_raw_data;
uint32_t press_raw_data;
};

extern PressureClass PressureShield; // Creation of external
FloatClass object

#endif // End of guard

```

A.2 PressureShield.cpp

```

#include "PressureShield.h" // Include header file

// Sensor library

void PressureClass::readCoefficients() {
    dig_T1 = ((uint16_t)((PressureShield.readRegister(BMP280_DIG_T1_MSB_REG)
        << 8) + PressureShield.readRegister(BMP280_DIG_T1_LSB_REG)));
    dig_T2 = ((int16_t)((PressureShield.readRegister(BMP280_DIG_T2_MSB_REG)
        << 8) + PressureShield.readRegister(BMP280_DIG_T2_LSB_REG)));
    dig_T3 = ((int16_t)((PressureShield.readRegister(BMP280_DIG_T3_MSB_REG)
        << 8) + PressureShield.readRegister(BMP280_DIG_T3_LSB_REG)));

    dig_P1 = ((uint16_t)((PressureShield.readRegister(BMP280_DIG_P1_MSB_REG)
        << 8) + PressureShield.readRegister(BMP280_DIG_P1_LSB_REG)));
    dig_P2 = ((int16_t)((PressureShield.readRegister(BMP280_DIG_P2_MSB_REG)
        << 8) + PressureShield.readRegister(BMP280_DIG_P2_LSB_REG)));
    dig_P3 = ((int16_t)((PressureShield.readRegister(BMP280_DIG_P3_MSB_REG)
        << 8) + PressureShield.readRegister(BMP280_DIG_P3_LSB_REG)));
}

```

```

dig_P4 = ((int16_t)((PressureShield.readRegister(BMP280_DIG_P4_MSB_REG)
    << 8) + PressureShield.readRegister(BMP280_DIG_P4_LSB_REG)));
dig_P5 = ((int16_t)((PressureShield.readRegister(BMP280_DIG_P5_MSB_REG)
    << 8) + PressureShield.readRegister(BMP280_DIG_P5_LSB_REG)));
dig_P6 = ((int16_t)((PressureShield.readRegister(BMP280_DIG_P6_MSB_REG)
    << 8) + PressureShield.readRegister(BMP280_DIG_P6_LSB_REG)));
dig_P7 = ((int16_t)((PressureShield.readRegister(BMP280_DIG_P7_MSB_REG)
    << 8) + PressureShield.readRegister(BMP280_DIG_P7_LSB_REG)));
dig_P8 = ((int16_t)((PressureShield.readRegister(BMP280_DIG_P8_MSB_REG)
    << 8) + PressureShield.readRegister(BMP280_DIG_P8_LSB_REG)));
dig_P9 = ((int16_t)((PressureShield.readRegister(BMP280_DIG_P9_MSB_REG)
    << 8) + PressureShield.readRegister(BMP280_DIG_P9_LSB_REG)));
}

void PressureClass::begin_config() {           //configuration -> do begin
    Wire.beginTransmission(BMP280_addr);
    Wire.write(0xF4);
    Wire.write(0b111111);   //temp -> full, press -> full, 20-bit (
        oversampling x 16), mode -> normal
    Wire.write(0xF5);
    Wire.write(0b0);       //standby time, filter
    Wire.endTransmission();
    delay(10);
}

void PressureClass::measure_config() {          //configuration -> do begin
    Wire.beginTransmission(BMP280_addr);
    Wire.write(0xF4);
    Wire.write(0b1111);    //temp -> full, press -> full, 20-bit (oversampling
        x 16), mode -> normal
    Wire.write(0xF5);
    Wire.write(0b1000);    //standby time, filter

    Wire.endTransmission();
    delay(10);
}

uint8_t PressureClass::readRegister(uint8_t RegSet) {
    uint8_t _result = 0;
    Wire.beginTransmission(BMP280_addr);
    Wire.write(RegSet);
    Wire.endTransmission(false);
    Wire.requestFrom(BMP280_addr, 1, true);
    _result = Wire.read();
    return _result;
}

uint32_t PressureClass::press_read() {           //value from sensor
    Wire.beginTransmission(BMP280_addr);         //sensor address
    Wire.write(0xF7);                          //first reg
}

```

```

Wire.endTransmission(false);
Wire.requestFrom(BMP280_addr, 3, true);
press_raw_data = Wire.read();
press_raw_data <= 8;
press_raw_data |= Wire.read();
press_raw_data <= 8;
press_raw_data |= Wire.read();
return press_raw_data;
}

uint32_t PressureClass::temp_read() { //value from sensor
    Wire.beginTransmission(BMP280_addr); //sensor address
    Wire.write(0xFA); //first reg
    Wire.endTransmission(false);
    Wire.requestFrom(BMP280_addr, 3, true);
    temp_raw_data = Wire.read();
    temp_raw_data <= 8;
    temp_raw_data |= Wire.read();
    temp_raw_data <= 8;
    temp_raw_data |= Wire.read();
    temp_raw_data >>= 4; //remove zeroes from last bit
    return temp_raw_data;
}

void PressureClass::readTemperature() { //temperature counting
    int32_t _var1, _var2;

    int32_t adc_T = temp_read();

    _var1 = (((adc_T >> 3) - ((int32_t)dig_T1 << 1)) * ((int32_t)dig_T2))
        >> 11;
    _var2 = (((((adc_T >> 4) - ((int32_t)dig_T1)) * ((adc_T >> 4) - ((int32_t)
        )dig_T1))) >> 12) * ((int32_t)dig_T3)) >> 14;

    t_fine = _var1 + _var2 ;
}

float PressureClass::readPressure() { //pressure counting
    int64_t _var1, _var2, _press;

    int32_t adc_P = PressureShield.press_read();
    adc_P >>= 4;

    _var1 = (t_fine) - 128000;
    _var2 = _var1 * _var1 * (int64_t)dig_P6;
    _var2 = _var2 + (_var1 * (int64_t)dig_P5) << 17);
    _var2 = _var2 + (((int64_t)dig_P4) << 35);
    _var1 = (_var1 * _var1 * (int64_t)dig_P3) >> 8) + (_var1 * (int64_t)
        dig_P2) << 12);
    _var1 = (((((int64_t)1) << 47) + _var1)) * ((int64_t)dig_P1) >> 33;
}

```

```

if (_var1 == 0) {
    return 0; // avoid exception caused by division by zero
}
_press = 1048576 - adc_P;
_press = (((_press << 31) - _var2) * 3125) / _var1;
_var1 = (((int64_t)dig_P9) * (_press >> 13) * (_press >> 13)) >> 25;
_var2 = (((int64_t)dig_P8) * _press) >> 19;

_press = (_press + _var1 + _var2) >> 8) + (((int64_t)dig_P7) << 4);
return (float)_press / 256; //pressure in Pa
}

//PressureShield library

void PressureClass::begin(void) {
#ifdef ARDUINO_ARCH_AVR
    #if SHIELDRELEASE == 1 // For shield version 1
        pinMode(PRESSURE_UPIN,OUTPUT); // Set pump pin
        pinMode(PRESSURE_RPIN,INPUT); // Set sensor pin
        pinMode(PRESSURE_MSPIN,INPUT); // Set manual switch pin
        digitalWrite(PRESSURE_UPIN,LOW); // Turn off pump
        analogReference(EXTERNAL);
    #endif
#endif
Wire.begin();
PressureShield.begin_config();
PressureShield.readCoefficients();
PressureShield.readTemperature(); //potrebne pre t_fine
delay(5000);
}

void PressureClass::calibration(void) { // Board
    calibration
    _minPressure=PressureShield.readPressure(); // Read pressure
    during initialization
    _maxPressure=105000.00; // Maximum sensor
    pressure
    _wasCalibrated=true;
}

float PressureClass::referenceRead(void) { // Reference read
    int val=analogRead(PRESSURE_RPIN); // Potentiometer value read
    float percento = map((float)val, 0.0, 1023.0, 0.0, 100.0); // Mapping potentiometer on 3.3V value to 0 - 100
    return percento;
}

void PressureClass::actuatorWrite(float aPercent) { // 
}

```

```

    Pump activator
float inPercento = aPercent*255/100;
analogWrite(PRESSURE_UPIN,inPercento); // Turn on pump
}

float PressureClass::sensorRead(void) { // Sensor read
    unsigned long int val=PressureShield.readPressure(); // Read sensor value
    if (val>105000) { // Sensor max value safety control
        digitalWrite(PRESSURE_UPIN,LOW);
    }
    float percento = map((long int)val, _minPressure,_maxPressure, 0.00,
        100.00); // Mapping sensor value to 0 - 100
    return percento;
}

PressureClass PressureShield; // Creation of PressureClass object

```

A.3 PressureShield_PID.ino

```

#include "PressureShield.h" // Include the library
#include <Sampling.h> // Include sampling

unsigned long Ts = 10; // Sampling in milliseconds
bool enable = false; // Flag for sampling
float r = 0.0;
float u = 0.0;
float y = 0.0;
float R[] = {80.0, 50.0, 70.0, 40.0, 60.0, 30.0, 50.0, 20.0, 40.0, 70.0,
    60.0};
int T = 500;
unsigned long k = 0;
int i;
bool MS; // Manual switch value

#define Kp 3 // PID Kp potom Kp=3
#define Ti 1 // PID Ti
#define Td 0.01 // PID Td

void setup() {

Serial.begin(2000000); // Initialize serial
PressureShield.begin(); // Initialization
PressureShield.calibration(); // Calibration

// Initialize sampling function
Sampling.period(Ts * 1000); // Sampling init.

```

```

Sampling.interrupt(stepEnable); // Interrupt fcn.

PIDAbs.setKp(Kp); // Proportional
PIDAbs.setTi(Ti); // Integral
PIDAbs.setTd(Td); // Derivative
PIDAbs.setTs(Sampling.samplingPeriod); // Sampling
}

void loop() {
    if (enable) { // If ISR enables
        step(); // Algorithm step
        enable=false; // Then disable
    }
}

void stepEnable() { // ISR
    enable=true; // Change flag
}

void step() {

MS = digitalRead(PRESSURE_MSPIN); // Manual switch monitoring

if (MS == 1) { // Manual switch set to Automatic mode

if (k % (T*i) == 0) {
    if (i==11) {
        i=0;
        k=0;
    }
    r = R[i]; // Set reference
    i++;
}

y = PressureShield.sensorRead(); // Read Sensor
u = PIDAbs.compute(r-y,0,100,0,100); // PID
PressureShield.actuatorWrite(u); // Actuate

k++;
}

if (MS == 0) { // Manual switch set to Manual mode

r = PressureShield.referenceRead(); // Read reference
y = PressureShield.sensorRead(); // Read Sensor
u = PIDAbs.compute(r-y,0,100,0,100); // PID
PressureShield.actuatorWrite(u); // Actuate
}

Serial.print(r); // Print reference
Serial.print(",_");
}

```

```

    Serial.print(PressureShield.sensorRead());      // Print output
    Serial.print(",");
    Serial.println(u);
}

```

A.4 PressureShield_LQ.ino

```

#include "PressureShield.h"
#include <Sampling.h>                                // Include interrupt-based sampling
                                                       framework

unsigned long Ts = 10;                                // Sampling period in milliseconds
unsigned long k = 0;                                  // Sample index
bool nextStep = false;                            // Flag for step function
bool realTimeViolation = false;                    // Flag for real-time sampling
                                                       violation

float R[] = {50.0, 70.0, 40.0, 60.0, 80.0, 60.0, 30.0, 70.0, 60.0};
float y = 0.0;
float u = 0.0;

int T = 1000;                                         // Section length
int i = 0;                                            // Section counter for pre-set reference

bool MS;

BLA::Matrix<2, 2> A = {0.993, 0.01526, -0.003223, 0.4206}; // State
                                                               matrix A
BLA::Matrix<2, 1> B = {0.008751, -0.1501};           // Input
                                                               matrix B
BLA::Matrix<1, 2> C = {1.74, 0.8755};                // Output
                                                               matrix C

// Kalman process and measurement error covariances
BLA::Matrix<2, 2> Q_Kalman = {0.05, 0, 0, 0.01};    // Process noise
                                                               covariance matrix
BLA::Matrix<1, 1> R_Kalman = {1};                     // Measurement noise
                                                               covariance matrix

// LQ gain with integrator
BLA::Matrix<1, 3> K = {1.25, 0.05, -0.01};          // Pre-
                                                               calculated LQ gain K
BLA::Matrix<3, 1> X = {0, 0, 0};                      // Estimated
                                                               state vector
BLA::Matrix<3, 1> Xr = {0, 0, 0};                     // Reference
                                                               state vector

void setup() {                                       // Setup - runs only once
    Serial.begin(2000000);                          // Begin serial communication
    PressureShield.begin();                         // Initialise PressureShield board
}

```

```

PressureShield.calibration();      // Calibrate PressureShield board

Sampling.period(Ts * 1000);        // Set sampling period in
                                 microseconds (Ts in ms)
Sampling.interrupt(stepEnable);   // Set interrupt function
}

void loop() {                      // Loop - runs indefinitely
    if (nextStep) {                // If ISR enables step flag
        step();                   // Run step function
        nextStep = false;          // Disable step flag
    }
}

void stepEnable() {                // ISR
    nextStep = true;              // Enable step flag
}

void step() {                      // Define step function
MS = digitalRead(PRESSURE_MSPIN);
if (MS==0) {                      // If Manual mode is active
    Xr(0) = PressureShield.referenceRead(); // Read reference from
                                              potentiometer
}
if (MS==1) {                      // If Automatic mode is
    active
    if (i > (sizeof(R) / sizeof(R[0]))) { // If at end of reference
        PressureShield.actuatorWrite(0.0); // Turn off the pump
        while (1);                     // Stop program execution
    } else if (k % (T * i) == 0) { // If at the end of section
        Xr(0) = R[i];               // Progress in reference
        i++;                         // Increment section
        counter
    }
}

y = PressureShield.sensorRead();   // Read sensor
PressureShield.getKalmanEstimate(X, u, y, A, B, C, Q_Kalman, R_Kalman);
// Estimate internal states X
X(2) = X(2) + (Xr(0) - X(0) - Xr(0)*0.35);
u = -(K * (X - Xr))(0);           // Calculate LQ
                                 system input
PressureShield.actuatorWrite(u);   // Actuate

Serial.print(Xr(0));              // Print reference
Serial.print(",");
Serial.print(y);                 // Print output
Serial.print(",");

```

```
Serial.println(u);  
  
    k++; // Increment index  
}
```

B MATLAB kód

B.1 PressureShield.m

```
% PressureShield MATLAB API.  
%  
% The script initializes the board, then reads the  
% output, writes to the input and reads reference  
% from the potentiometer.  
%  
% This code is part of the AutomationShield hardware and software  
% ecosystem. Visit http://www.automationshield.com for more  
% details. This code is licensed under a Creative Commons  
% Attribution-NonCommercial 4.0 International License.  
%  
% Created by Martin Staron.  
% Last update: 4.5.2021.  
  
classdef PressureShield < handle  
  
    properties(Access = public)  
        arduino;  
        dev;  
    end  
  
    properties(Constant)  
        Pressure_RPIN = 'A0'; % BMP280 (Sensor)  
        Pressure_UPIN = 'D11'; % Pump (Actuator)  
    end  
  
    methods  
        function begin(PressureShieldObject)  
            PressureShieldObject.arduino = arduino();  
            PressureShieldObject.dev = device(PressureShieldObject.arduino,  
                'I2CAddress','0x76');  
            BMP280Init(PressureShieldObject.dev);  
            disp('PressureShield_initialized.')  
        end  
  
        function actuatorWrite(PressureShieldObject, percent)  
            writePWMDutyCycle(PressureShieldObject.arduino,  
                PressureShieldObject.Pressure_UPIN, (percent/100));
```

```

    end

    function reference = referenceRead(PressureShieldObject)
        reference = readVoltage(PressureShieldObject.arduino,
            PressureShieldObject.Pressure_RPIN) * 100 / 5;
    end

    function y = sensorRead(PressureShieldObject)
        y = BMP280(PressureShieldObject.dev);
    end

end

```

B.2 BMP280Init.m

```

% MATLAB function to initialize temperature and pressure
% sensor BMP280
%
% This code is part of the AutomationShield hardware and software
% ecosystem. Visit http://www.automationshield.com for more
% details. This code is licensed under a Creative Commons
% Attribution-NonCommercial 4.0 International License.
%
% Created by Martin Staron.
% Last update: 26.4.2021.

function y = BMP280Init(dev),
    writeRegister(dev,'F4',0b1111);
    writeRegister(dev,'F5',0b1000);
end

```

B.3 BMP280.m

```

% MATLAB function to read temperature and pressure
% from BMP280 temperature and pressure sensor.
%
% This code is part of the AutomationShield hardware and software
% ecosystem. Visit http://www.automationshield.com for more
% details. This code is licensed under a Creative Commons
% Attribution-NonCommercial 4.0 International License.
%
% Created by Martin Staron.
% Last update: 26.4.2021.

function y = BMP280(dev)
persistent i;
global dig_T1 dig_T2 dig_T3 dig_P1 dig_P2 dig_P3 dig_P4 dig_P5 dig_P6
dig_P7 dig_P8 dig_P9 adc_t t_fine;
if (isempty(i))
    i = 1
end

```

```

end

if (i==1)

    UT_MSB = readRegister(dev,'FA','uint8');
    UT_LSB = readRegister(dev,'FB','uint8');
    UT_XLSB = readRegister(dev,'FC','uint8');

    UT_MSBbit = bitget(UT_MSB,8:-1:1);
    UT_LSBbit = bitget(UT_LSB,8:-1:1);
    UT_XLSBbit = bitget(UT_XLSB,8:-1:5);
    bitsT = [UT_MSBbit UT_LSBbit UT_XLSBbit];
    adc_t = bi2de(bitsT,'left-msb');

    dig_T1M = readRegister(dev,'89','uint8');
    dig_T1L = readRegister(dev,'88','uint8');
    dig_T1Mbit = bitget(dig_T1M,8:-1:1);
    dig_T1Lbit = bitget(dig_T1L,8:-1:1);
    bitsT1 = [dig_T1Mbit dig_T1Lbit];
    dig_T1 = bi2de(bitsT1,'left-msb');

    dig_T2M = readRegister(dev,'8B','uint8');
    dig_T2L = readRegister(dev,'8A','uint8');
    dig_T2Mbit = bitget(dig_T2M,8:-1:1);
    dig_T2Lbit = bitget(dig_T2L,8:-1:1);
    bitsT2 = [dig_T2Mbit dig_T2Lbit];
    dig_T2 = bi2de(bitsT2,'left-msb');

    dig_T3M = readRegister(dev,'8D','uint8');
    dig_T3L = readRegister(dev,'8C','uint8');
    dig_T3Mbit = bitget(dig_T3M,8:-1:1);
    dig_T3Lbit = bitget(dig_T3L,8:-1:1);
    bitsT3 = [dig_T3Mbit dig_T3Lbit];
    dig_T3 = bi2de(bitsT3,'left-msb');

    dig_P1M = readRegister(dev,'8F','uint8');
    dig_P1L = readRegister(dev,'8E','uint8');
    dig_P1Mbit = bitget(dig_P1M,8:-1:1);
    dig_P1Lbit = bitget(dig_P1L,8:-1:1);
    bitsP1 = [dig_P1Mbit dig_P1Lbit];
    dig_P1 = bi2de(bitsP1,'left-msb');

    dig_P2M = readRegister(dev,'91','uint8');
    dig_P2L = readRegister(dev,'90','uint8');
    dig_P2Mbit = bitget(dig_P2M,8:-1:1);
    dig_P2Lbit = bitget(dig_P2L,8:-1:1);
    bitsP2 = [dig_P2Mbit dig_P2Lbit];
    dig_P2 = bi2de(bitsP2,'left-msb');

    dig_P3M = readRegister(dev,'93','uint8');
    dig_P3L = readRegister(dev,'92','uint8');

```

```

dig_P3Mbit = bitget(dig_P3M,8:-1:1);
dig_P3Lbit = bitget(dig_P3L,8:-1:1);
bitsP3 = [dig_P3Mbit dig_P3Lbit];
dig_P3 = bi2de(bitsP3,'left-msb');

dig_P4M = readRegister(dev,'95','uint8');
dig_P4L = readRegister(dev,'94','uint8');
dig_P4Mbit = bitget(dig_P4M,8:-1:1);
dig_P4Lbit = bitget(dig_P4L,8:-1:1);
bitsP4 = [dig_P4Mbit dig_P4Lbit];
dig_P4 = bi2de(bitsP4,'left-msb');

dig_P5M = readRegister(dev,'97','uint8');
dig_P5L = readRegister(dev,'96','uint8');
dig_P5Mbit = bitget(dig_P5M,8:-1:1);
dig_P5Lbit = bitget(dig_P5L,8:-1:1);
bitsP5 = [dig_P5Mbit dig_P5Lbit];
dig_P5 = bi2de(bitsP5,'left-msb');

dig_P6M = readRegister(dev,'99','uint8');
dig_P6L = readRegister(dev,'98','uint8');
dig_P6Mbit = bitget(dig_P6M,8:-1:1);
dig_P6Lbit = bitget(dig_P6L,8:-1:1);
bitsP6 = [dig_P6Mbit dig_P6Lbit];
dig_P6 = bi2de(bitsP6,'left-msb');

dig_P7M = readRegister(dev,'9B','uint8');
dig_P7L = readRegister(dev,'9A','uint8');
dig_P7Mbit = bitget(dig_P7M,8:-1:1);
dig_P7Lbit = bitget(dig_P7L,8:-1:1);
bitsP7 = [dig_P7Mbit dig_P7Lbit];
dig_P7 = bi2de(bitsP7,'left-msb');

dig_P8M = readRegister(dev,'9D','uint8');
dig_P8L = readRegister(dev,'9C','uint8');
dig_P8Mbit = bitget(dig_P8M,8:-1:1);
dig_P8Lbit = bitget(dig_P8L,8:-1:1);
bitsP8 = [dig_P8Mbit dig_P8Lbit];
dig_P8 = bi2de(bitsP8,'left-msb');

dig_P9M = readRegister(dev,'9F','uint8');
dig_P9L = readRegister(dev,'9E','uint8');
dig_P9Mbit = bitget(dig_P9M,8:-1:1);
dig_P9Lbit = bitget(dig_P9L,8:-1:1);
bitsP9 = [dig_P9Mbit dig_P9Lbit];
dig_P9 = bi2de(bitsP9,'left-msb');

var_1 = ((adc_t)/16384-(dig_T1)/1024)*(dig_T2);
var_2 = (((adc_t)/131072-(dig_T1)/8192)*((adc_t)/131072-(dig_T1
    )/8192))* (dig_T3);
t_fine = var_1 + var_2;

```

```

    i=2
end

UP_MSB = readRegister(dev,'F7','uint8');
UP_LSB = readRegister(dev,'F8','uint8');
UP_XLSB = readRegister(dev,'F9','uint8');

UP_MSBbit = bitget(UP_MSB,8:-1:1);
UP_LSBbit = bitget(UP_LSB,8:-1:1);
UP_XLSBbit = bitget(UP_XLSB,8:-1:5);
bitsP = [UP_MSBbit UP_LSBbit UP_XLSBbit];
adc_p = bi2de(bitsP,'left-msb');

var1 = (t_fine/2)-64000;
var2 = var1*var1*dig_P6/32768;
var2 = var2+var1*dig_P5*2;
var2 = (var2/4)+(dig_P4*65536);
var1 = (dig_P3*var1*var1/524288+dig_P2*var1)/524288;
var1 = (1+var1/32768)*dig_P1;
p = 1048576-adc_p;
p = (p-(var2/4096))*6250/var1;
var1 = dig_P9*p*p/2147483648;
var2 = p*dig_P8/32768;
y = int32((p+(var1+var2+dig_P7)/16));

end

```

B.4 PressureShield_PID.m

```

% PressureShield simple PID example
%
% Performs a discrete PID control of the PressureShield
% hardware with input saturation.
%
% This example initializes the PressureShield device, then
% requests a step change of reference to 50 hPa.
% The input to the pump is calculated by an absolute
% discrete PID controller in its ideal form. The inputs are saturated,
% no integral windup handling is provided. The example plots the live
% response. When the experiment is over, the data is saved to a file.
%
% This code is part of the AutomationShield hardware and software
% ecosystem. Visit http://www.automationshield.com for more
% details. This code is licensed under a Creative Commons
% Attribution-NonCommercial 4.0 International License.
%
% Created by Martin Staron.
% Last update: 26.4.2021.

startScript;

```

```

PressureShield = PressureShield;                                % Construct object from class
PressureShield.begin();                                       % Initialize shield

Kp = 0.9;                                                 % PID Gain
Ti = 1;                                                   % PID Integral time constant
Td = 0.1;                                                 % PID Derivative time constant

umin = 0;                                                 % Minimum input
umax = 100;                                              % Maximum input

R=[60 40 70 50 80];                                     % [HPa] Overpressure closed-
loop reference

Ts = 0.1;                                                 % [s] Sampling period
runTime = 75;                                              % [s] Total runtime

secLength = 100;                                           % Length of a reference trajectory section
stepEnable = 0;                                            % Algorithm step flag
k = 1;                                                    % Algorithm step counter
j = 1;                                                    % Reference section counter
r = R(1);                                                 % First reference
eSum = 0;                                                 % Error sum
ep = 0;                                                   % Previous error
y = 0;                                                    % Output initialize
init = PressureShield.sensorRead();                         % Read reference pressure

tic                                                       % Start measuring time
while(1)                                                 % Infinite loop
    if (stepEnable)&&(toc>5.0)                           % If flag is enabled
        % Read the output
        y = (PressureShield.sensorRead()-init)/100;      % [HPa] OverPressure

        % Pick reference
        if (mod(k,secLength*j)==0);                      % If time for new reference
            j=j+1;                                         % Next reference section
            if (j == 6)
                j = 1;
            end
            r=R(j);                                       % Pick new reference
        end

        % Compute PID
        e = r-y;                                         % [HPa]
        Reference
        eSum = eSum+e;                                    % Integral
        eSum = constrain((Kp*Ts/Ti)*eSum,umin,umax)/(Kp*Ts/Ti);
        u = double(Kp*(e+(Ts/Ti)*eSum+(Td/Ts)*(e-ep))); % PID
        ep = e;                                           % Data
        store
        u = constrain(u,umin,umax);                      % Input
        saturation

```

```

% Write to hardware
PressureShield.actuatorWrite(u); % [%] Power
response(k,:) = [r y u]; % Store results
%plotLive(response(k,:)); % Live plot

k = k+1; % Next sample no.
stepEnable = 0; % Disable step.
end % Step end
if (toc<runTime) % If its time
    stepEnable = 1; % Enable the step
elseif (toc>=runTime) % Experiment over
    PressureShield.actuatorWrite(0); % Input off
    break % Exit while
end
end

save response response % Data file with response
plot(response)

```

B.5 PressureShield_LQ_Simulation.m

```

startScript;

Ts=0.01; %sampling period

load('Reference.mat')
load('BlackBoxModel.mat')

dsystem=disSystem2;
A=dsystem.A;
B=dsystem.B;
C=dsystem.C;
D=dsystem.D

%integrator
AI = [A, zeros(2, 1); -C, 1];
BI = [B; 0];
CI = [C, 0];

%penalization matrices
R=100;
Q=diag([1 10 0.01]);

[K, P]= dlqr(AI,BI,Q,R); %LQ gain calculation

Ref=Reference;
Ref=Ref(1:1500)';

N=length(Ref);
t=1:N;

```

```

x0=[0.01; 0];
X=X0;
R=Ref;
XI = 0;

%simulation
for k=1:N

    U(k)=-K*[X(:,k); XI];

    if U(k)>100
        U(k)=100;
    elseif U(k)<0
        U(k)=0;;
    end;

    X(:,k+1)=A*X(:,k)+B*U(k);
    Y(:,k)=C*X(:,k);
    XI = XI + (R(k)-Y(k));
end
K
figure(1)
subplot(211)

plot(t,R,'b',t,Y,'r')
title('Output_Y')

xlabel('Time_(s)')
subplot(212)
plot(t,U,'r');
title('Input_U')
xlabel('Time_(s)')

```

B.6 PressureShield_LQ.m

```

% PressureShield closed-loop LQ response example
%
% LQ feedback control of overpressure in the PressureShield.
%
% This example initialises and calibrates the board
% and starts linear quadratic (LQ) control
% using a pre-defined reference. At the end
% the results are stored in 'responseLQ.mat' file and are shown on the
% figure on the screen.
%
% This code is part of the AutomationShield hardware and software
% ecosystem. Visit http://www.automationshield.com for more
% details. This code is licensed under a Creative Commons

```

```

% Attribution-NonCommercial 4.0 International License.
%
% Created by Martin Staro?.
% Last update: 16.5.2021.

startScript; % Clears command window, variables and opened
    figures
clear estimateKalmanState; % Clears persistent variables in estimate
    function

PressureShield = PressureShield; % Create PressureShield object from
    PressureShield class
PressureShield.begin; % Initialises shield

Ts = 0.2; % Sampling period in seconds
k = 1; % Algorithm step counter
nextStep = 0; % Algorithm step flag
samplingViolation = 0; % Sampling violation flag
init = PressureShield.sensorRead();

Ref = [50 70 40 60 80 60 30 70 60 50]; % Reference overpressure in hPa
T = 160; % Section length
i = 0; % Section counter
response = zeros(length(Ref)*T, 3); % Preallocate output variable

umin = 0;
umax = 100;

X = [0; 0; 0]; % State vector
Xr = [0; 0; 0]; % Reference vector

% System state-space matrices

matA = [0.993, 0.01526; -0.003223, 0.4206];
matB = [0.008751; -0.1501];
matC = [1.74, 0.8755];

% Penalisation matrices
Q_kalman = [0.05, 0; 0, 0.59]; % State penalisation
R_kalman = 1; % Input penalisation

% Calculate LQ gain that includes integrator state
K = [1.2501, 0.05, -0.01];
init = PressureShield.sensorRead();

tic % Start measuring time
while (1) % Infinite loop
    if (nextStep) % If step flag is enabled
        if (mod(k, T*i) == 1) % At the end of section, progress in
            reference
            i = i + 1;

```

```

if (i > length(Ref))      % If at the end of reference
    PressureShield.actuatorWrite(0.0);
    break                      % Stop the program execution
end
Xr(1) = Ref(i);            % Progress in reference
end
y = double((PressureShield.sensorRead()-init)/100);    % Read
overpressure
u = -K * (X - Xr);          % Calculate LQ system input
u = constrain(u,umin,umax);
PressureShield.actuatorWrite(u);                % Actuate

X(1:2) = estimateKalmanState(u, y, matA, matB, matC, Q_kalman,
R_kalman);
X(3) = X(3) + (Xr(1) - X(1) - Xr(1)*0.36);
response(k, :) = [Xr(1), y, u];           % Store results
k = k + 1;                                % Increment step counter
nextStep = 0;                               % Disable step flag
end                                         % Step end

if (toc>=800)
    samplingViolation = 1      % Set sampling violation flag
    PressureShield.actuatorWrite(0);
    break                      % Stop program if they do overlap
end
nextStep = 1;                                % Enable step flag
end
response = response(1:k-1, :);        % Remove unused space
save responseLQ response             % Save results in responseLQ.mat file
plotResults('responseLQ.mat')        % Plot results from responseLQ.mat file

```

B.6 PressureShield_MPC_Simulation.m

```

startScript;

load('Reference.mat')
load('BlackBoxModel.mat')

dsystem=disSystem2;
A=dsystem.A;
B=dsystem.B;
C=dsystem.C;
D=dsystem.D;

%integrator
AI = [A, zeros(2, 1); -C, 1];
BI = [B; 0];
CI = [C, 0];

Np = 10;
% Constraints

```

```

uL = 0;           % Lower input limit
uU = 100;         % Upper input limit

%penalization matrices
R=5;
Q=diag([1 0.001 1]);

[K, P] = dlqr(AI, BI, Q, R);
% Get Hessian, Gradient and F matrices of cost function
[H, G, F] = ucelovafunkcia(AI, BI, Np, Q, R, P);
% Set input constraints onto the system
[Ac, b0] = obmedzenia(uL, uU, Np);

H = (H + H') / 2;           % Create symmetric Hessian matrix
opt = optimoptions('quadprog', 'Display', 'none'); % Turn off display

Ref=Reference;
Ref=Ref(1:3000)';

N=length(Ref);
t=1:N;

X0=[0.01; 0];
X=X0;
R=Ref;
XI = 0;

%simulation
for i = 1:N
    Xn(:,i)=[X(:,i);XI]; %update extended state vector
    u_opt = quadprog(H,G*Xn(:,i),Ac,b0,[],[],[],[],opt); %calculate
        optimal input with integrator
    U(i) = u_opt(1); %select input for each step
    X(:,i+1) = A*X(:,i) + B*U(i); %the "real" system (no integrator
        included)
    XI = XI + (Ref(i)-X(1,i)); % update integrator state
end

figure(1)
subplot(2,1,1)
plot(X(1,:))
hold on
plot(Ref)
axis([0,3000,0,100])
ylabel('Pretlak_(hPa)')
xlabel('Vzorka')
subplot(2,1,2)
plot(U)
axis([0,3000,0,100])
ylabel('PMW_(%)')
xlabel('Vzorka')

```

B.6 PressureShield_MPC.m

```
% Model Predictive Control (MPC) feedback control of overpressure
% in the PressureShield.

%
% This example initialises and calibrates the board
% and starts Model Predictive Control (MPC)
% using a pre-defined reference. At the end
% the results are stored in 'responseLQ.mat' file and are shown on the
% figure on the screen.

%
% This code is part of the AutomationShield hardware and software
% ecosystem. Visit http://www.automationshield.com for more
% details. This code is licensed under a Creative Commons
% Attribution-NonCommercial 4.0 International License.

%
% Created by Martin Staron.
% Last update: 29.5.2021.

startScript; % Clears screen and variables
clear estimateKalmanState;

PressureShield = PressureShield; % Create PressureShield object from
PressureShield class
PressureShield.begin; % Initialises shield

Ts = 0.15;
k = 1;
nextStep = 0;
samplingViolation = 0;

Ref = [40 50 40 60 40 50 60 70 60 50]; % Reference overpressure in hPa

T = 750; % Section length
i = 0; % Section counter
response = zeros(length(Ref)*T, 3); % Preallocate output variable

X = [0; 0; 0]; % State vector
Xr = [0; 0; 0]; % Reference vector

% System state-space matrices
matA = [0.993, 0.01526; -0.003223, 0.4206];
matB = [0.008751; -0.1501];
matC = [1.74, 0.8755];

matAhat = [matA, zeros(2, 1); -matC, 1];
matBhat = [matB; 0];

% Constraints
uL = 0; % Lower input limit
uU = 100; % Upper input limit
```

```

% Penalisation matrices and horizon
R=8;
Q=diag([1 10 0.01]);
np = 2; % Prediction horizon

Q_Kalman = [0.01, 0; 0, 0.51]; % State penalisation
R_Kalman = 15;
runTime = 1000;

% Get final state penalisation matrix P
[K, P] = dlqr(matAhat, matBhat, Q, R);
% Get Hessian and Gradient matrices of cost function
[H,G]=getCostFunctionMPC(matAhat,matBhat,np,Q,R,P);
[Ac b0]=setConstraintsMPC(uL,uU,np);

H = (H + H') / 2;
opt = optimoptions('quadprog', 'Display', 'none');
init = PressureShield.sensorRead();

tic % Start measuring time
while (1) % Infinite loop
    if (nextStep) % If step flag is enabled
        if (mod(k, T*i) == 1)
            i = i + 1;
            if (i > length(Ref))
                PressureShield.actuatorWrite(0.0);
                break
            end
            Xr(1) = Ref(i);
        end
        y = double((PressureShield.sensorRead()-init)/100); % Read overpressure
        % Predict system inputs
        u(:, 1) = quadprog(H, G*X(:, 1), Ac, b0, [], [], [], [], [], opt);
        PressureShield.actuatorWrite(u(1, 1)); % Actuate
        % Estimate states with Kalman filter
        X(1:2) = estimateKalmanState(u(1, 1), y, matA, matB, matC, Q_Kalman
            , R_Kalman);
        X(3) = X(3) + (Xr(1) - X(1) - Xr(1)*0.38);
        response(k, :) = [Xr(1), y, u(1, 1)]; % Store results
        k = k + 1;
        nextStep = 0; % Disable step flag
    end

    if (toc>=runTime)
        samplingViolation = 1 % Set sampling violation flag
        PressureShield.actuatorWrite(0);
        break % Stop program if they do overlap
    end
    nextStep = 1; % Enable step flag

```

```
end
toc
response = response(1:k-1, :); % Remove unused space
save responseMPC response % Save results in responseMPC.mat file
plotResults('responseMPC.mat') % Plot results from responseMPC.mat
file
```